42nd EuroForth Conference

on Forth and Concatenative Languages

(On-Premise Proceedings)

Cardiff, Wales
September 4-6, 2026

Preface

EuroForth is an annual conference on the Forth programming language, concatenative languages, stack machines, and related topics, and has been held since 1985. While concatenative languages have always been on-topic (Manfred von Thun presented Joy at EuroForth 2001), we decided to make this more explicit in order to widen the potential submitters and the audience.

The 42nd EuroForth finds us in Cardiff; in 2025 it was held in Hamburg, and in 2024 in Newcastle upon Tyne. Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

Since 1994, EuroForth has a refereed and a non-refereed track. This year there has been one submissions to the refereed track, and it was accepted. Since 2006, there have been 33 submissions, 24 accepts, 73% acceptance rate. The reviews of all papers are anonymous to the author: All papers were reviewed and the final decision taken without involving the author, including the submission authored by a program-committee member. I thank the program committee for their paper reviews and the authors for their submissions.

Several papers were submitted to the non-refereed track in time to be included in these on-premise proceedings. Late papers as well as those slides that will be submitted to the editor are included in the post-conference proceedings. The proceedings website `http://www.euroforth.org/ef26/papers/` also contains links to videos of the presentations when they are available. I thank the authors for their papers. In addition to the papers and presentation handouts available before the conference, these online proceedings also contain papers and presentation handouts that were provided at or after the conference. Also, some of the papers included in the printed proceedings were updated for these online proceedings. I thank the authors for their papers and slide handouts.

You can find these proceedings, as well as the individual papers and (when they become available) slides and links to the presentation videos on `http://www.euroforth.org/ef25/papers/`.

Workshops and social events complement the program. This year's EuroForth is organized by Nick and Janet Nelson.

Anton Ertl

Program committee

M. Anton Ertl, TU Wien (chair)
Marcel Hendrix
Matthias Koch
Jaanus Pöial, Tallinn University of Technology
Bradford Rodriguez, T-Recursive Technology
Friedrich Steimann, Fernuniverstität Hagen
Bill Stoddart
Reuben Thomas

Contents

Refereed Papers
Jaanus Pöial: Configurable Static Type Checking for Forth 5
Non-Refereed Papers
Axel Prütz Kanne and Robert Glück: Towards Reversible Stack-Based Computing 17
Nick J. Nelson: Does AI speak Forth? 23
Nick J. Nelson: Autotranslation within a Forth application — two different approaches 36
Nick J. Nelson: Enhancements to a Forth style enumeration word set . 49
Nick J. Nelson: SNIPPET: and INSERT — named source snippets in VFX Forth 53
Bill Stoddart: The problem of false positives in statistical tests: countering mistaken intuitions with Monte Carlo simulations 59
Bill Stoddart: Russell's Paradox Disarmed 64
Presentation Slides
François Laagel: Implementing Exceptions in Z79Forth/A 71
M. Anton Ertl: Gforth 1.0 77
M. Anton Ertl: Control Structure Enhancements 80
M. Anton Ertl: Floating-point output 81
Stephen Pelc: Security and ARM64 83

Configurable Static Type Checking for Forth

Jaanus Pöial
Tallinn University of Technology

Abstract

Forth stack comments document the cells consumed and produced by a word. Ordinary comments are not checked and do not record how stack words move or copy individual cells. This paper presents a configurable source-level static type checker implemented in gforth. It describes each word by a typed input/output stack contract, called a stack effect. A validation profile defines the type hierarchy and primitive word effects (specifications). Symbolic indices record the flow introduced by stack shuffling words such as `DUP`, `SWAP`, and `OVER`.

The analysis uses three algebraic operations. Composition handles linear word sequences. Meet operation unifies alternative branches. A finite one-pass-versus-two-pass comparison handles repetition. Forth examples provided in the paper illustrate these operations. They cover type refinement, cell identity, branch merging, reusable inferred effects, and local loop stability. The prototype derives every reported effect without executing concrete values. Its conclusions depend on the user-defined type hirarchy, primitive specifications and source text.

Keywords: Forth; static type checking; stack effects; symbolic execution.

1 Introduction

A Forth word is a named operation. Here, a cell is one single-cell stack item. The data stack is the ordered sequence of those items. A word normally receives operands from that stack and returns its results to the same stack. The conventional notation

```
+   ( n1 n2 -- sum )
```

places the top of stack on the right and says that `+` consumes two numbers and produces one. Such comments are valuable interfaces, but a Forth system usually consumes them as source text rather than treating them as proof obligations. A comment can therefore be missing, stale, or semantically stronger than the checked definition.

Consider the familiar definition

```
: SQUARED  ( n -- n^2 )  DUP * ;
```

The checker can derive `( n - n )` from the declarations of `DUP` and `*`. It can verify the number and identity of the cells used by the phrase. It does not prove the mathematical assertion that the result is the square of the input.

The checker reads three text inputs:

1. a file that defines types, aliases, and ordering relations (type hierarchy);
2. a file that defines word effects and source-reading rules;
3. a Forth source file whose phrases are checked and whose colon definitions are inferred.

A *primitive word* is a word whose effect comes directly from the second file. A *colon definition* begins with `:` and ends with `;`. The word that it defines receives an inferred effect from the source

between those delimiters. A top-level phrase is checked and traced but does not add a dictionary entry. The prototype consumes the three files and maps each known word to one stack effect. At `;`, the inferred body effect is added to this dictionary. The prototype analyzes symbolic stack cells; it does not execute the Forth definition. The inferred definitions are stored in a log file.

The design builds on earlier work on stack effects and symbolic type variables [3, 4, 5, 6]. The following topics are discussed:

- how a later word can constrain an earlier input;
- how two branch effects are combined;
- why an effect displays only the top cells that it touches;
- why the loop check compares one pass with two passes.

Sections 2–4 define the analysis used here. Sections 5–7 derive all twelve test results and the four helper effects.

2 Typed Symbolic Stack Effects

2.1 Types and compatibility

Let T be a finite set of abstract cell types with a precision order $\preceq$. The relation $t \preceq u$ means that t is more specific than u. A value of type t may therefore be used where u is required. This order is the profile's subtype relation. The Forth-2012-like profile used for the examples contains, among other edges, the transitive chains

$$\texttt{a-addr} \preceq \texttt{c-addr} \preceq \texttt{addr} \preceq \texttt{u} \preceq \texttt{n} \preceq \texttt{x},$$

$$\texttt{char} \preceq \texttt{+n} \preceq \texttt{u} \preceq \texttt{n} \preceq \texttt{x}, \qquad \texttt{flag} \preceq \texttt{x}.$$

The flag type is deliberately separate from the numeric chain in this profile. Another profile may classify flags as numbers. The hierarchy is therefore a policy for one target vocabulary. It is not a universal claim about Forth cells. Names such as `x`, `X`, and `cell` may be aliases for one type.

Two types are *comparable* when one is below the other in $\preceq$. Comparable types are compatible. Matching them retains the more specific type:

$$\operatorname{match}(t,u) = \begin{cases} t, & t \preceq u, \\ u, & u \preceq t, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

The prototype does not search for a third common subtype when t and u are incomparable. An undefined match is a type clash.

2.2 Symbols, effects, and the implicit frame

A symbolic cell is a type with an optional positive identity index, written $t[i]$. Equal indices denote the same abstract origin. Different indices do not claim that the concrete values differ. Every unindexed occurrence starts as an independent symbol. An unindexed output is therefore a fresh abstract result. Indices exist only in the checker. They are not addresses or runtime tags.

The declarations of three stack words illustrate the notation:

```
DUP   ( x[1] -- x[1] x[1] )
SWAP  ( x[2] x[1] -- x[1] x[2] )
OVER  ( x[2] x[1] -- x[2] x[1] x[2] )
```

Before each word occurrence is analyzed, its local indices receive fresh numbers. Two primitive calls therefore cannot share an index by accident. After analysis, *normalization* renumbers the surviving indices from one upward. Renumbering itself changes only their printed names. The prototype also compacts identities: a derived index that records only an implicit one-to-one correlation may be hidden, while an explicitly declared or multiply occurring correlation is retained. Once hidden, that pairwise correlation is unavailable to later composition. This deliberate precision loss keeps stored effects small but means that inferred effects need not be principal.

An effect $e = (L - -R)$ contains two finite symbolic stacks, with their tops on the right. A stack *suffix* is the group of top cells displayed by the effect. Deeper cells form an implicit *frame*. For every frame F, the contract acts as

$$F\,L \longmapsto F\,R.$$

For example, `( a-addr - x )` for `@` does not require the entire data stack to contain one cell. It requires only that the top cell be an aligned address; any deeper F passes through unchanged. The identity effect is `( - )`. Write ϵ for the empty stack. A clash is a separate failure result and must not be confused with the identity effect. The set of symbolic effects is the checker's *abstract domain*. An effect operation is *partial* when some arguments produce no effect. In those cases, this checker reports a clash.

3 Main Effect Operations

3.1 Sequential composition

Write typed symbolic cells by juxtaposition, with the top on the right, and let e and f denote possibly empty sequences of such symbols. The two symbolic equations from the presentation capture the basic cases:

$$(a - -b) \cdot (e\,b - -d) = (e\,a - -d), \qquad (a - -f\,c) \cdot (c - -d) = (a - -f\,d).$$

In the first equation, the second effect consumes b but also needs the deeper sequence e; that sequence therefore becomes part of the composite input. In the second, the first effect leaves f below c; consuming c leaves f below the new result d. In case of type mismatch the result is zero (type clash).

More generally, $p = (L_p - -R_p)$ followed by $q = (L_q - -R_q)$ is written $p \cdot q$. Boundary cells are matched from the top inward. Comparable types are replaced by their more specific type, and that refinement is substituted through every occurrence of the same symbolic index. Thus a late `@` requirement can propagate through copies made earlier by `SWAP` and `DUP`. Incomparable boundary types make composition undefined. A sequence is composed from left to right, and the empty sequence has the identity effect. This operation extends earlier untyped stack-effect composition [3].

3.2 Common branch contracts

Alternative paths use the common-contract operation $p \sqcap q$. It produces one must-style interface rather than a set of path effects. Its two basic symbolic equations are

$$(a - -b) \sqcap (e\,a - -e\,b) = (e\,a - -e\,b),$$

$$(e\,a\,--e\,b) \sqcap (e\,c\,--e\,d) = \big(e\ \min(a,c)\ --e\ \max(b,d)\big).$$

The first equation makes the implicit frame explicit: $(a--b)$ also acts as $(e\,a--e\,b)$, so that is the common contract. In the second equation, $\min(a,c)$ means the more specific of two comparable input types, while $\max(b,d)$ means the more general output type. These are precision-order operations, not numeric minimum and maximum. Inputs narrow because a caller must satisfy both paths; outputs widen because a client may assume only what both paths guarantee.

Positions are aligned from the top. An identity index survives only when both paths preserve the same input-to-output relationship; a path-dependent permutation is forgotten. Unequal displayed depths are valid only when the extra input and output depths are equal, because the omitted prefix must pass through unchanged:

$$|L_p| - |L_q| = |R_p| - |R_q|.$$

A different net stack change or an incomparable aligned type makes the meet undefined (zero).

Write $[\![P]\!]$ for the inferred effect of a source segment P. The selector is composed separately for a conditional:

$$[\![\texttt{IF}\ \ P\ \ \texttt{ELSE}\ \ Q\ \ \texttt{FI}]\!] = (\texttt{flag} --) \cdot ([\![P]\!] \sqcap [\![Q]\!]).$$

Without `ELSE`, Q is the identity effect.

3.3 Finite repetition

For a repeated phrase with one-pass effect p, the prototype computes

$$\pi_2(p) = p \sqcap (p \cdot p),$$

provided both the composition and common-contract operations are defined. This one-versus-two comparison is finite and terminating. It detects many bodies that grow, shrink, or change the stack incompatibly on a second pass.

The notation has a narrow meaning. It does not summarize every possible iteration count. One and two passes can agree after information has been lost. A later pass may then remove another correlation. A full loop analysis would propagate the loop-header state until that state stops changing. π_2 does not perform that process. It also proves neither termination nor properties of concrete values. It is only a local stack-stability test.

4 Prototype Inputs

The prototype loads a type profile, a word-and-control profile, and the example source. Each example below introduces its primitive declarations at the point of use rather than repeating them in a separate catalogue.

Control syntax is also profile-driven. A conditional captures its two branches and composes the selector with their common contract. A `BEGIN-WHILE-REPEAT` form captures a prefix and a body, while `BEGIN-UNTIL` captures one body including the final selector; the configured recipe applies π_2 to those segments. The profile therefore determines both how source is divided and which symbolic operation combines each part.

With these profiles, the prototype accepts all twelve tests and the four helper definitions. Their inferred effects are derived once, in the following sections. The accepted top-level phrase is discussed with `test12`.

5 Straight-Line Examples

5.1 Test1: stack shape without value evaluation

```
: test1 3 4 5 - + ;
```

Every literal has effect `( - n )`. Naming fresh abstract results a, b, and c, the stack evolves as

$$\epsilon \longrightarrow n_a \longrightarrow n_a\ n_b \longrightarrow n_a\ n_b\ n_c \longrightarrow n_a\ n_d \longrightarrow n_e.$$

Subtraction consumes n_b, n_c and produces a fresh n_d. Addition consumes n_a, n_d and produces n_e. No input is required. One number is left:

$$\texttt{test1}\ (--n).$$

Concrete execution would calculate $4 - 5 = -1$ and then $3 + (-1) = 2$. The checker does not perform that calculation. Its domain contains the type `n`, but no integer constants or arithmetic expressions. It establishes only `( - n )`.

5.2 Test2: a generic shuffle specialized to flags

```
: test2 OR FALSE SWAP ;
```

`OR` first requires two flags and leaves one fresh flag. `FALSE` adds another flag. The generic inputs of `SWAP` then match these two results. Both inputs are refined from `x` to `flag`:

$$\texttt{flag}_a\ \texttt{flag}_b \xrightarrow{\texttt{OR}} \texttt{flag}_c \xrightarrow{\texttt{FALSE}} \texttt{flag}_c\ \texttt{flag}_d \xrightarrow{\texttt{SWAP}} \texttt{flag}_d\ \texttt{flag}_c.$$

The inputs are consumed by `OR`; neither output is correlated with an original input. The reported interface therefore needs no visible indices:

$$\texttt{test2}\ (\texttt{flag flag - flag flag}).$$

Although one output is produced by the word named `FALSE`, the checker has no constant-propagation domain and promises only its declared type.

5.3 Test3: backward refinement through `SWAP` and `DUP`

```
: test3 SWAP DUP @ ;
```

Let the initial suffix be $A\ B$, with B topmost. `SWAP` produces $B\ A$ and `DUP` produces $B\ A\ A$. The final `@` requires the top copy of A to be an `a-addr`. Because both copies carry the same symbolic index, the substitution also refines the untouched copy and the original input:

$$A\ B \xrightarrow{\texttt{SWAP}} B\ A \xrightarrow{\texttt{DUP}} B\ A\ A \xrightarrow{\texttt{@}} B\ A\ X_{\text{fresh}}.$$

After normalized numbering, A is `a-addr[2]` and B is `x[1]`. Hence

$$\texttt{test3}\ (\texttt{a-addr[2] x[1]}\ \texttt{- x[1] a-addr[2] x}).$$

This is the key identity-flow example. Tracking only stack depth would miss the address requirement. Tracking types without indices would also lose information. Such an analysis could refine only the consumed copy. It could not identify the surviving copy as the same aligned address.

6 Branch Examples

6.1 Test4: an explicit effect versus an implicit frame

```
: test4 IF 1 + FI ;
```

The true branch `1 +` pushes an `n`, then needs one additional `n` below it for `+`. Its inferred effect is `( n - n )`. The absent `ELSE` branch has the identity effect `( - )`. These displayed depths differ, but the empty branch preserves an arbitrary deeper frame. Making one frame cell explicit gives the alignment

$$\begin{array}{rcl} \text{true} & : & (\texttt{n - n}), \\ \text{false} & : & (\underline{\texttt{n}} - -\underline{\texttt{n}}). \end{array}$$

Both paths preserve a one-cell numeric stack shape. They do not preserve the same value. The false path returns the original number. The true path returns the result of addition. No output identity is claimed. After the leading `IF` consumes its selector, the complete definition is

$$\texttt{test4}\ (\texttt{n flag - n}).$$

This example explains why comparing only the explicitly printed sides would be too strict. Stack effects denote transformations under an implicit frame.

6.2 Test5: a common prefix survives, a branch-dependent identity does not

```
: test5 IF DUP ELSE 0 FI ;
```

The true branch is `( x[1] - x[1] x[1] )`. The false branch explicitly has `( - n )`, but it leaves the deeper frame unchanged. Exposing one frame cell aligns the branches as follows:

$$\begin{array}{rcl} \text{true} & : & (\texttt{x[1] - x[1] x[1]}), \\ \text{false} & : & (\texttt{x[1] - x[1] n}). \end{array}$$

The lower output is the original input on both paths, so its index survives. The top output is the same input on the true path but a fresh numeric literal on the false path. Its path-independent type is the more general type `x`. It cannot retain the input index. The branch contract is thus

$$(\texttt{x[1] - x[1] x}).$$

Composing the selector yields

$$\texttt{test5}\ (\texttt{x[1] flag - x[1] x}).$$

The result loses two facts. First, `n` is widened to `x` at the differing output. Second, a correlation valid on only one path is discarded.

6.3 Test6: compatible shape with no common permutation

```
: test6 IF ROT ELSE @ FI ;
```

Write the pre-branch suffix as $A\ B\ C$, with C topmost. The true branch rotates it:

$$A\ B\ C \longmapsto B\ C\ A.$$

The false branch `@` displays only one input and one output. Its implicit two-cell frame makes the comparable transformation

$$A\ B\ C \longmapsto A\ B\ D,$$

where C must be `a-addr` and D is a fresh `x`. At the input, `ROT`'s generic requirement and `@`'s address requirement meet at the more specific `a-addr`. The input contract is consequently `( x x a-addr - ... )`.

The output positions have the path pairs (B, A), (C, B), and (A, D). None of these establishes one common source identity. Their only common output type is `x`, so the branch contract is

$$(\texttt{x x a-addr - x x x}).$$

After selector consumption, the inferred definition is

$$\texttt{test6}\ (\texttt{x x a-addr flag - x x x}).$$

The checker accepts the equal net stack change. It forgets which input reaches which output. An analysis that stores each path separately could retain both permutations. This checker stores one common contract.

6.4 Test9: input and output variance in isolation

```
: test9 IF @ ELSE C@ FI ;
```

The two branch effects are

$$(\texttt{a-addr - x}) \quad \text{and} \quad (\texttt{c-addr - char}).$$

An input to the whole conditional must satisfy both address requirements. Since $\texttt{a-addr} \preceq \texttt{c-addr}$, the common input is the more specific `a-addr`. An output client, however, needs a guarantee valid for either `x` or `char`. Since $\texttt{char} \preceq \texttt{x}$, that guarantee is the more general `x`. Thus

$$(\texttt{a-addr - x}) \sqcap (\texttt{c-addr - char}) = (\texttt{a-addr - x}),$$

and selector composition gives

$$\texttt{test9}\ (\texttt{a-addr flag - x}).$$

This small example is the clearest demonstration of contract variance: branch inputs are narrowed, whereas outputs are widened.

6.5 Test10: merging effects inferred for helper words

```
: P INVERT HERE ! DP ! DP @ DP C@ SWAP ;
: Q SWAP ! DP @ DP ;
: test10 IF P ELSE Q FI ;
```

This example first checks two ordinary colon definitions and then uses their inferred effects as branch contracts. For `P`, let the initial suffix be `x flag`. `INVERT HERE !` consumes the new aligned address and the flag result, leaving the lower `x`. `DP !` then consumes that cell and another aligned address. Finally, `DP @` and `DP C@` produce `x char`, which `SWAP` exchanges. Thus

$$\texttt{P}\ (\texttt{x flag - char x}).$$

Both outputs are fresh results of fetches; neither is an original input.

For `Q`, `SWAP !` consumes an `a-addr` and an arbitrary cell. `DP @ DP` then produces a fresh cell and a fresh aligned address:

$$\texttt{Q}\ (\texttt{a-addr x - x a-addr}).$$

The common branch input narrows `x` against `a-addr` in the lower position and `flag` against `x` in the upper position. The two output pairs are `(char,x)` and `(x,a-addr)`; each position therefore widens to `x`. No identity is common to the paths:

$$[\![\texttt{P}]\!] \sqcap [\![\texttt{Q}]\!] = (\texttt{a-addr flag - x x}).$$

After `IF` consumes the selector, the complete result is

$$\texttt{test10}\ (\texttt{a-addr flag flag - x x}).$$

Unlike `test9`, this case shows that effects inferred for user-defined words are immediately reusable and that variance is applied independently at each aligned position.

7 Repetition Examples

7.1 Test7: one-versus-two exposes both flag requirements

```
:  test7 BEGIN SWAP OVER WHILE INVERT REPEAT ;
```

The profile divides this structure into the repeated prefix `SWAP OVER WHILE` and the repeated body `INVERT`. Consider one prefix pass on $A\ B$:

$$A\ B \xrightarrow{\texttt{SWAP}} B\ A \xrightarrow{\texttt{OVER}} B\ A\ B \xrightarrow{\texttt{WHILE}} B\ A.$$

`WHILE` consumes the duplicate of the original top cell, so B must be a flag. A second prefix pass transforms $B\ A$ back to $A\ B$ and tests A. Consequently the two-pass effect requires both original cells to be flags. The common summary preserves two flag-typed positions. It cannot promise one permutation. One pass swaps the cells, while two passes restore them. The summary is therefore

$$(\texttt{flag flag - flag flag}).$$

The second repeated region, `INVERT`, has the stable type effect `( flag - flag )`. Composing it with the prefix summary leaves the same two-cell interface, giving

$$\texttt{test7}\ (\texttt{flag flag - flag flag}).$$

This is a useful rejection test for changing stack shape and incompatible second-pass types. It is not a proof that the reported identity-free summary characterizes every number of loop iterations or that the loop terminates.

7.2 Test8: a data-stack-neutral loop pass

```
:  test8 BEGIN KEY 0= UNTIL ;
```

A complete repeated pass has the data-stack trace

$$\epsilon \xrightarrow{\texttt{KEY}} \texttt{char} \xrightarrow{\texttt{0=}} \texttt{flag} \xrightarrow{\texttt{UNTIL}} \epsilon.$$

The loaded `x` contract for `0=` accepts `char` directly. The presentation's narrower `n` annotation would also accept it because $\texttt{char} \preceq \texttt{n}$. `UNTIL` consumes the resulting flag. One pass has the identity data-stack effect. Two passes have the same effect. Their common contract is

$$\texttt{test8}\ (\text{-}).$$

This conclusion concerns only the modeled data stack. `KEY` still has an I/O side effect. The result does not promise a key that meets the exit condition. It also does not prove termination.

7.3 Test11: a reusable body refines both loop cells

```
:  BODY SWAP DUP @ 0= ;
:  test11 BEGIN BODY UNTIL ;
```

The first three words of `BODY` repeat the identity flow from `test3`. The fetch result is a generic `x`; the loaded profile's broad `( x - flag )` declaration for `0=` consumes it and produces a selector. The helper effect is

$$\texttt{BODY}\ (\texttt{a-addr[2] x[1] - x[1] a-addr[2] flag}).$$

`UNTIL` consumes the final flag. One complete pass therefore swaps two cells and requires the first one to be an aligned address:

$$p = (\texttt{a-addr[2] x[1] - x[1] a-addr[2]}).$$

On a second pass, the other original cell reaches `@` and is refined to `a-addr`. Hence $p \cdot p$ requires two aligned addresses and restores their order. One pass swaps the origins while two passes preserve them, so no input/output identity is common to both paths. Their types are common:

$$p \sqcap (p \cdot p) = (\texttt{a-addr a-addr - a-addr a-addr}).$$

This is the reported effect of `test11`. It combines reusable inferred identity information with the same two-pass refinement principle seen in `test7`.

7.4 Test12: a concrete boundary of one-versus-two

```
:  BODY23 ROT INVERT DROP 0 ;
:  test12 BEGIN BODY23 KEY? UNTIL ;
BODY23 BODY23 BODY23
```

Let the three inputs of `BODY23` be A B C, with C topmost. `ROT` produces B C A, so `INVERT` requires A to be a flag. `DROP` removes the resulting flag, and the literal produces a fresh `n`. The reusable effect is

$$\texttt{BODY23}\ (\texttt{flag x[2] x[1] - x[2] x[1] n}).$$

In `test12`, `KEY?` produces the flag consumed by `UNTIL`; those two words have no net data-stack effect. Thus one complete loop pass has the same effect as `BODY23`. A second pass requires B to be a flag and maps A B C to C n n. Meeting one and two passes retains the requirements that A and B are flags, widens the first two output positions to `x`, and retains `n` at the top:

$$\texttt{test12}\ (\texttt{flag flag x - x x n}).$$

No index survives because the first output originates at B after one pass but at C after two, while the remaining aligned outputs also lack a common origin.

The final top-level phrase deliberately takes a third pass. Its symbolic trace is

$$\texttt{flag flag flag} \longrightarrow \texttt{flag flag n} \longrightarrow \texttt{flag n n} \longrightarrow \texttt{n n n}.$$

The third pass introduces a requirement not present in the one-versus-two summary: C must also be a flag. This does not contradict the rule, because π_2 was defined as a bounded local comparison, not as an inductive loop invariant. It gives a concrete reason not to read the accepted `test12` contract as valid for every iteration count.

8 Discussion

8.1 What the examples establish

The twelve tests isolate complementary properties of the symbolic domain. `test1` shows that the checker infers shape and cell type without constant evaluation. `test2` shows subtype specialization of a generic stack word. `test3` demonstrates bidirectional information flow: execution order is left to right, but a late requirement propagates backward through shared symbolic identities.

The branch examples show why an effect is more than a pair of stack heights. `test4` depends on the implicit frame. `test5` retains only an identity common to both paths. `test6` loses a branch-dependent permutation. `test9` isolates input/output variance, and `test10` applies that variance to effects inferred for helper words. The loop examples expose both the value and the limit of the finite rule. A second pass refines another cell in `test7`; a balanced pass becomes identity in `test8`; and `test11` carries identity information through a reusable body. Finally, `test12` and the following three-`BODY23` phrase show directly why one-versus-two is not an all-iteration invariant.

All reported outputs are inferred from primitive contracts. The illustrative comments in the presentation are not declarations. This rule prevents a stale comment from changing inference. Semantic labels such as `value` or "address contents" are therefore outside the domain. A richer profile would be needed to represent them.

8.2 Configuration as an analysis parameter

The same source may receive a different verdict under another valid profile. For example, one profile may classify `char` as numeric. It accepts the displayed `KEY 0=`. Another profile may keep `char` and `n` unrelated. A numeric-only `0=` then rejects the phrase. Even under the hierarchy used here, replacing the loaded broad `( x - flag )` contract by `( n - flag )` would still accept `test8` but reject `BODY`, because `@` guarantees only `x`. The target Forth system determines which policy is appropriate. The profile makes that policy reviewable.

External control declarations provide similar flexibility. They also enlarge the trusted base. A recipe might omit a selector effect. It might also put `UNTIL` outside its repeated segment. Either error can produce a wrong summary even when the three operations are correct. The configuration should therefore be reviewed with the checked code.

9 Validity and Limitations

Acceptance has a precise but bounded interpretation:

- every checked linear boundary has enough modeled cells of comparable types;
- every checked branch admits one common contract under the defined alignment and variance rules;
- every checked repetition admits the finite $p \sqcap (p \cdot p)$ summary;
- parser roles, delimiters, and compilation state fit the active profile.

It does not imply full Forth program correctness. The principal threats to a stronger interpretation are as follows.

Trusted declarations. Primitive and parser specifications are assumptions. If `@`, `KEY`, or a control word is declared incorrectly, analysis of its clients cannot make the result reliable.

Single effects, compaction, and comparable types. The dictionary stores one effect per word. Branch merging also returns one common contract. It cannot retain a separate permutation for each path. Normalization may discard an implicit one-to-one identity, so a stored derived effect need not retain every correlation available in its construction trace. Aligned types must be directly comparable. The checker does not search for a third common type.

Finite loop approximation. The one-versus-two comparison does not prove a result for every iteration count. A stronger analysis would maintain an abstract stack at the loop header. It would propagate the body until that stack stopped changing. Another design would require a declared loop interface and check every back edge against it [1].

Forth language coverage. Return-stack manipulation, unrestricted `EXECUTE`, recursion, nonlocal `THROW`, native defining words, self-modifying dictionaries, and unrestricted metaprogramming require additional abstractions. The checker should complement runtime tests, code review, and target-system knowledge.

The examples are explanatory cases, not a performance benchmark or a proof that every accepted program is safe. Each case makes one operation visible, and the prototype's reported effects agree with `examples.txt.log`.

10 Related Work

Forth has long used stack-effect notation as an interface convention. The Forth standard defines data types and control semantics [2]. It does not make each source comment a static judgment. Earlier work developed algebraic composition and multiple effects for control flow [3, 4]. Later work added typed symbolic variables and conservative control-flow operations [5, 6]. The prototype described here also loads parser and control behavior from a profile.

Stoddart and Knaggs also studied type inference for stack languages [8]. Saabas and Uustalu gave compositional type systems for stack-based low-level code [7]. Forth source can additionally contain parser words and dictionary extensions. The profile handles that variability, but it becomes part of the trusted input.

Abstract interpretation offers a stronger treatment of loops [1]. It propagates abstract states until they stop changing. The one-versus-two rule performs only the first bounded comparison. An inferred loop effect must be read with that distinction in mind.

11 Conclusion

Typed symbolic stack effects turn a familiar Forth notation into executable, configurable documentation. Sequential composition checks touching stack boundaries and propagates refined types through shared identities. Branch merging computes one conservative contract, narrowing inputs, widening outputs, and preserving only path-independent correlations. Repetition compares one pass with two to obtain a useful but explicitly finite stability summary.

The tests provided make these mechanisms concrete. Three initial linear definitions show type and identity propagation. Five conditionals show implicit frames, input narrowing, output widening, information loss, and reuse of inferred helper contracts. Four loops show what the bounded repetition rule accepts; the last is paired with a three-pass phrase that exposes exactly what the rule does not establish. The prototype derives every reported effect under the selected profile. It is a practical checker for the configured subset of Forth. It is not a proof of functional behavior or termination. Stronger assurance requires validated primitives, path-sensitive alternatives, less lossy identity handling, and full loop analysis.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL*, pp. 238–252, 1977. doi:10.1145/512950.512973. Public full text: `https://www.di.ens.fr/~cousot/publications.www/CousotCousot-POPL-77-ACM-p238--252-1977.pdf`.

[2] Forth 200x Standardisation Committee. *Forth 2012 Standard*. Draft proposed standard, 10 November 2014. Public full text: `https://forth-standard.org/standard/intro`.

[3] J. Pöial. Algebraic specification of stack effects. *Forth Dimensions*, 16(4):18–20, 1994. Public full text: `https://www.forth.org/fd/FD-V16N4.pdf`.

[4] J. Pöial. Multiple stack-effects of Forth programs. In *1991 FORML Conference Proceedings, euroFORML'91*, pp. 400–406, 1992. Public full text: `https://www.complang.tuwien.ac.at/anton/euroforth/ef91/poial.pdf`.

[5] J. Pöial. Stack effect calculus with typed wildcards, polymorphism and inheritance. Presentation and proceedings abstract, 18th EuroForth Conference, 2002. Public full text: `https://www.complang.tuwien.ac.at/anton/euroforth/ef02/poial02.pdf`.

[6] J. Pöial. Typing tools for typeless stack languages. In *Proceedings of the 22nd EuroForth Conference*, pp. 40–46, 2006. Public full text: `https://www.complang.tuwien.ac.at/anton/euroforth/ef06/poial06.pdf`.

[7] A. Saabas and T. Uustalu. Compositional type systems for stack-based low-level languages. In *Proceedings of CATS 2006*, CRPIT 51, pp. 27–39, 2006. Public full text: `https://web.archive.org/web/20120325100058id_/http://crpit.com/confpapers/CRPITV51Saabas.pdf`.

[8] B. Stoddart and P. J. Knaggs. Type inference in stack based languages. In *1991 FORML Conference Proceedings, euroFORML'91*, pp. 407–421, 1991. Public full text: `https://www.complang.tuwien.ac.at/anton/euroforth/ef91/stoddart.pdf`.

Towards Reversible Stack-Based Computing

Axel Prütz Kanne & Robert Glück

aj72kanne@gmail.com

Department of Computer Science, University of Copenhagen, Denmark

Introduction

Reversible computing is a field of study concerned with computing systems in which computation is deterministic in both the forward and backward directions and therefore information preserving [1]. One of the main motivations for this field is the promise of reduced power consumption. *Landauer's Principle*, proposed by Rolf Landauer in 1961 and later experimentally verified, states that any loss of information in a closed system leads to dissipation of energy [2]. Reversible computers could therefore potentially circumvent this physical limit using reversible circuitry.

Achieving this would require both hardware and software to be created with reversibility in mind. Reversible architectures proposed have mainly been register-based, such as the *Pendulum* [3,4] and *Bob* [5] architectures, and higher-level reversible programming languages have also been designed, like the imperative language Janus [6,7], functional RFUN [8,9] and object-oriented ROOPL [10].

However another interesting idea would be to combine stack-based and reversible computing, and here some work has already been accomplished:

- *RVM-Forth* a weakly reversible virtual machine for running Forth [11,12].
- The *Wrongforth* language, a reversible version of Forth [13].
- A stack machine with a clean instruction set has also been designed [14].

This work seeks to expand the field by exploring the potential of combining reversible and stack-based computing, both on a lower level by proposing a reversible stack-based instruction set architecture called *To-And-Fro*, and on a higher level with a design for a reversible concatenative language inspired by Forth called *Do-Si-Do*, which compiles to To-And-Fro.

To-And-Fro

Abstract Machine Design

The To-And-Fro stack architecture aims to be simple in terms of instruction set and hardware resources. This makes it potentially suitable for embedded systems with limited resource requirements, such as battery life. To achieve this, the proposed architecture has the following properties:

- Pure stack addressing: Operands are not part of the instruction format, and no general-purpose registers are used. This gives a very compact instruction format, with no need to describe operands, resulting in a simpler instruction decoding unit.
- Minimal instruction set: The instruction set for the machine aims to be minimal while still being powerful enough to be useful. This further simplifies instruction decoding and reduces the required hardware resources
- Cleanly reversible: The reversibility of the machine does not rely on reverbilization methods such as Landauer Embedding [15], which uses program tracing. Instead, instructions are *reversible updates* of the machine state [16], where each instruction has an inverse counterpart that implements the inverse semantics.

The abstract machine enables this by using both classical and reversible components:

- **Primary Stack**: This is a hardware stack containing the stack elements that most instructions will operate on. This stack is a collection of registers that works as a cyclic buffer.

- **Secondary Stack**: This is also a hardware stack, used either to temporarily hold elements from the primary stack or to hold return addresses.
- **Program Counter Register** (*pc*): This special-purpose register contains the address of the current instruction being executed.
- **Direction Bit Register** (*dir*): This special-purpose register indicates the current execution direction: 1 represents the forward execution direction, whereas −1 represents backward execution. Additionally, the current instruction's inverse semantics will be executed if the direction bit is −1.
- **Branch Register** (*br*): This special purpose register holds relative jump values, which are the only form of jump allowed. The program counter is updated as follows:
  - $br = 0 \Rightarrow pc \mathrel{+}= dir$
  - $br \neq 0 \Rightarrow pc \mathrel{+}= br \cdot dir$

  By saving the value of the relative jump in the branch register, it is always possible to determine the address of the previous instruction.
- **General Purpose Memory**: Memory unit used for general purpose data.
- **Program Memory**: To-And-Fro uses a *Harvard architecture*, in which the program has its own memory unit. Because instructions cannot modify the program memory, it is not possible to overwrite the current instruction, which can lead to irreversibility.

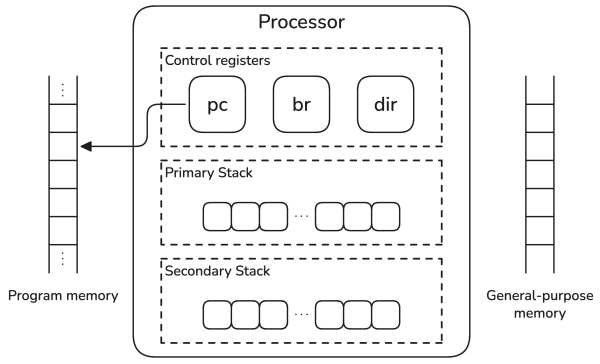


Figure 1: The abstract machine.

Both general purpose memory and the stack registers are zero-valued initially.

Instruction Set Architecture

Because the *push* and *pop* operations of a conventional stack machine are inherently irreversible due to information loss, the stacks of this machine work by doing a *rotation* of all stack elements using the `PUSH0` and `POP0` instructions. `PUSH0` retrieves the bottom element of the primary stack and places it at the top, while `POP0` does the inverse, placing the top element at the bottom of the stack.

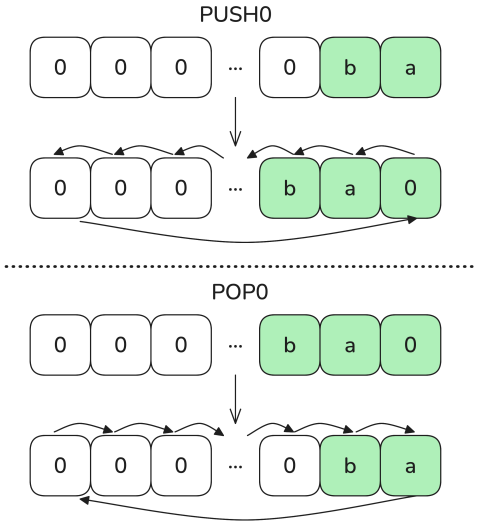


Figure 2: Rotation of stack elements in the the `PUSH0` and `POP0` instructions.

The main idea is to keep the invariant of a *clean stack* by using `POP0` only when the top element is zero-valued. This ensures that the top element after `PUSH0` will always be zero-valued, assuming there is no stack overflow.

The traditional push and pop operations can then be achieved by pairing `PUSH0` and `POP0` with an `XORI` instruction. The `XORI` instruction updates the top element of the primary stack using bit-wise exclusive-or with an immediate value. The following snippet shows an example where the value 2 is pushed to the stack, and later popped:

```
tnf
PUSH0   ;   -- 0
XORI 2  ; 0 -- 2
...     ; other code
XORI 2  ; 2 -- 0
POP0    ; 0 -- 2
```

Figure 3: Code showing how to push a value to the stack and pop it afterwards. The notation `0 -- 2` shows the contents of the primary stack before (`0`) and after (`2`) the instruction has been executed.

Another example of instructions that are not reversible on a conventional stack machine, is addition and subtraction of stack elements. On a conventional stack machine this is done by popping the two top elements of the stack and pushing back the result. But since neither addition nor subtraction is injective, this is an irreversible operation. To achieve reversibility, the `ADD` and `SUB` instructions of To-And-Fro leave the second operand on the stack unchanged. This means that the previous state can be reconstructed simply by executing the inverse, since addition and subtraction are injective in their first argument [16].

Table 1 shows all instructions that operate exclusively on the primary stack. One other notable instruction is `SIGN`, which is used to bit-wise negate the top element if the second element is negative. This is easy to check hardware-wise, since one only needs to look at the sign-bit of the second element. By combining `SIGN` with the `SUB` instruction, it is possible to compare stack elements and produce a *flag* value indicating the result of the comparison.

Inst	Stack Effect	Inv(Inst)
PUSH0	-- 0	POP0
POP0	0 --	PUSH0
XORI imm	n -- n^imm	XORI imm
ADD	b a -- b (a+b)	SUB
SUB	b a -- b (a-b)	ADD
SWAP	b a -- a b	SWAP
INC	n -- n+1	DEC
DEC	n -- n-1	INC
RL1	n -- n <<< 1	RR1
RR1	n -- n >>> 1	RL1
NOT	n -- ~n	NOT
SIGN	n c -- n (n<0 ? ~c : c)	SIGN

Table 1: Table showing instructions of To-And-Fro which only operate on the primary stack. Here ^ is bit-wise exclusive-or, ~ is bit-wise negation, <<< is bit-wise rotation left and >>> right.

There are also instructions that modify elements other than those on the primary stack, which can be seen in Table 2:

- `MEM` is used to perform a memory swap, using the second element as the target address and the top as the value to swap with.
- `TS` and `FS` are used to move an element *to* the secondary stack or retrieve an element *from* the secondary stack.
- `IF off` this is used for conditional jumps by adding an offset to the branch register if the top element is negative.
- `SBR` and `SRBR` are used for relative jumps by swapping the branch register and the top element. `SRBR` also flips the direction bit. These instructions are used to perform calls to reversible procedures.

Inst	Effect	Inv(Inst)
MEM	P: adr e -- adr M[adr] M[adr] := e	MEM
TS	P: e -- S: -- e	FS
FS	P: -- e S: e --	TS
IF off	P: c -- c br += (c<0) ? off : 0	IF -off
SBR	P: off -- br br := off	SBR
SRBR	P: off -- br br := off dir *= -1	SRBR

Table 2: Table showing instructions of To-And-Fro which do not only change the primary stack.

Listing 1 shows a To-And-Fro program, in which the greatest of the two top of elements x and y is duplicated. This is done in the following way:

- Lines 2-7: The two top elements are compared for less-than equality, and the resulting flag is placed on top of the stack. This flag represents the truth value of $x < y$.
- Lines 9-24: The `IF 5` instruction on line 9 is used for choosing between two branches of code depending on the flag, and thereby creates a reversible if-else control-flow:
  - Lines 10-13 ($x \geq y$): The flag was false (0). x is duplicated and a jump is made to the paired `IF` instruction on line 24.
  - Lines 14-23 ($x < y$): The flag was true (-1). The branch register has the value 5 on line 14, so the `IF -5` instruction will zero-clear the branch register, and the program resumes sequential execution afterwards. Lines 17-22 duplicates the second top element, while keeping the original ordering the same. The final line `PUSH0` will push a false flag on top of the stack to fall through the conditional jump on line 24.
  - Line 24: This `IF` instruction is the join point of the conditional control-flow. However, the flag value now represents the truth value of $x \geq y$.
- Lines 26-35: The garbage flag value representing $x \geq y$ is removed in a similar way to how the initial flag value was created on lines 2-7.

This small instruction set of just 18 instructions, is powerful enough for implementing non-trivial programs. Because of the lower number of instructions, it is not easy to use directly, but acts best as a compilation target for a higher-level language.

tnf

```
 1         ; y x
 2 SUB     ; y (x-y)
 3 PUSH0   ; y (x-y) 0
 4 SIGN    ; y (x-y) (x<y)
 5 TS      ; y (x-y)
 6 ADD     ; y x
 7 FS      ; y x (x<y)
 8
 9 IF 5    ; jump if (x<y)
10 ADD     ; y x x
11 PUSH0   ; y x x 0
12 NOT     ; y x x -1
13 IF 11   ; jump
14 IF -5   ; clear
15 NOT     ; y x 0
16 POP0    ; y x
17 SWAP    ; x y
18 PUSH0   ; x y 0
19 ADD     ; x y y
20 TS      ; x y
21 SWAP    ; y x
22 FS      ; y x y
23 PUSH0   ; y x y 0
24 IF -11  ; clear if (x>=y)
25
26 NOT     ; y x max (x<y)
27 SWAP    ; y x (x<y) max
28 TS      ; y x (x<y)
29 TS      ; y x
30 SUB     ; y (x-y)
31 FS      ; y (x-y) (x<y)
32 SIGN    ; y (x-y) (x<y ^ x<y)
33 POP0    ; y (x-y)
34 ADD     ; y x
35 FS      ; y x max
```

Listing 1: A To-And-Fro program, where the greatest of the two top stack elements x and y is duplicated.

Do-Si-Do

Do-Si-Do[1] is a higher-level, stack-based reversible language inspired by Forth that builds on the To-And-Fro instruction set. Do-Si-Do provides constructs such as conditionals, loops and word definitions directly. The program shown in Listing 1 can be rewritten as the following `MAX_DUP` word:

did

```
1 / y x -- y x max
2 : MAX_DUP
3     IF <= FI
4       DUP       / y x x
5     THEN
6       [SWAP]1   / y x y
7     ELSE -ROT <= EIF
```

Figure 4: The Do-Si-Do version of the To-And-Fro program shown in Listing 1.

The above code uses an if-construct, with the *test sequence* `<=` creating the *test flag*, and the *exit sequence* `-ROT <=` creating the *exit flag*. Reversible control-flow can be implemented using a test condition to choose between the branches, and an exit assertion to assert which branch was executed [1]. However, when a Do-Si-Do if-construct is compiled to To-And-Fro instructions, the exit sequence is used to remove the garbage flag value. Changes done during production of flags in if and loop constructs are reverted afterwards.

One unique construct of Do-Si-Do is the *setup-copy-revert* (SCR) construct, which is an adaptation of the compute-uncompute method, also colloquially known as Bennet's trick [17]. In Listing 1 `[SWAP]1` swaps the order of `x` and `y`, copies `y`, swaps them back to the original order, and places the copy of `y` on top, thereby achieving the same as the `OVER` word in Forth. The construct has the form `[SETUP]n(EFFECT)`, which has five phases:

1. The *setup sequence* `SETUP` is called to arrange the stack in a specific way.
2. The *effect sequence* `EFFECT` is called to perform a side effect.
3. The top `n` elements are copied.
4. `SETUP` is uncalled, to revert the changes done by the setup-sequence.
5. The copied elements are placed on the stack.

This is useful in many cases where a temporary stack setup is needed to produce a specific value, or where a setup is needed for doing a specific side effect.

To illustrate how reversible loops and SCR constructs work, Figure 5 shows the word definition for a reversible version of Fibonacci:

did

```
1 / n -- fib(n+1) fib(n)
2 : FIBPAIR
3     1 1
4     FR = OM
5       [ROT](-1) + SWAP
6     LOOP ROT =0 UNTIL
7     ROT ~0
```

Figure 5: The Fibonacci pair word definition, adapted from [1].

This word calculates the `n`th Fibonacci number using an iterative approach, by maintaining a *pair* of the current and next Fibonacci number:

- Line 3: The initial pair (1,1) is pushed to the stack.
- Line 4: The loop *entry sequence* `=` is used to test the two top elements for equality. The flag produced should be *true* when the loop is entered, and *false* afterwards.
- Line 5: `[ROT](-1)` is an SCR construct which temporarily places the input `n` on top to decrement it. Afterwards, `+ SWAP` calculates the next pair.
- Line 6: The *test sequence* `ROT =0` produces a flag value to determine when to exit the loop. If the flag is true, the loop is exited, and else the entry sequence is executed and the loop is repeated.
- Line 7: This sequence is used to clean up the input value which is now 0. If `~` is prefixed to a word, the inverse semantics is executed, so `~0` is the same as popping 0.

[1] *Do-si-do* is a dance move where a pair of dancers pass each other back-to-back without turning, and return to their initial spot: https://en.wikipedia.org/wiki/Do-si-do.

Bibliography

[1] R. Glück, T. Yokoyama, Reversible computing from a programming language perspective, Theoretical Computer Science 953 (2023) 113429.

[2] M. Krakovsky, Taking the heat, Commun. ACM 64 (2021) 18–20.

[3] M. Frank, Reversibility for Efficient Computing, Doctoral dissertation, Massachusetts Institute of Technology, 1999.

[4] C.J. Vieri, Reversible computer engineering and architecture, Doctoral dissertation, Massachusetts Institute of Technology, 1999.

[5] M.K. Thomsen, H.B. Axelsen, R. Glück, A Reversible Processor Architecture and Its Reversible Logic Design, in: A. De Vos, R. Wille (Eds.), Reversible Computation, Springer Berlin Heidelberg, 2012: pp. 30–42.

[6] C. Lutz, Janus: a time-reversible language, (1986).

[7] T. Yokoyama, R. Glück, A reversible programming language and its invertible self-interpreter, in: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Association for Computing Machinery, 2007: pp. 144–153.

[8] M.K. Thomsen, H.B. Axelsen, Interpretation and programming of the reversible functional language RFUN, in: Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, Association for Computing Machinery, 2015.

[9] T. Yokoyama, H.B. Axelsen, R. Glück, Towards a reversible functional language, in: A. De Vos, R. Wille (Eds.), Reversible Computation, Springer, Switzerland, 2012: pp. 14–29.

[10] T. Haulund, T.Æ. Mogensen, R. Glück, Implementing reversible object-oriented language features on reversible machines, in: P. Iain, R. Hafizur (Eds.), Reversible Computation. Proceedings, Springer-Verlag, 2017: pp. 66–73.

[11] B. Stoddart, RVM-FORTH, a Reversible Virtual Machine, in: Proceedings of the European Forth Conference (Euroforth), EuroForth Conference, 2004.

[12] B. Stoddart, R. Lynas, F. Zeyda, A Virtual Machine for Supporting Reversible Probabilistic Guarded Command Languages, Electronic Notes in Theoretical Computer Science 253 (2010) 33–56.

[13] G. Wodni, Wrongforth - A reversible Forth, in: Proceedings of the European Forth Conference (Euroforth), 2009: pp. 42–45.

[14] N. Deworetzki, U. Meyer, Designing a Reversible Stack Machine, in: C.A. Mezzina, K. Podlaski (Eds.), Reversible Computation, Springer International Publishing, 2022: pp. 65–72.

[15] R. Landauer, Irreversibility and heat generation in the computing process, IBM J. Res. Dev. 5 (1961) 183–191.

[16] H.B. Axelsen, R. Glück, T. Yokoyama, Reversible Machine Code and Its Abstract Processor Architecture, in: V. Diekert, M.V. Volkov, A. Voronkov (Eds.), Computer Science – Theory and Applications, Springer Berlin Heidelberg, 2007: pp. 56–69.

[17] C.H. Bennett, Logical Reversibility of Computation, IBM Journal of Research and Development 17 (1973) 525–532.

EuroForth 2026

Does AI speak Forth?

Abstract

Can an AI assistant write Forth? Much better than one would expect — once it has been taught your house rules. This paper reports a measured 5.8-fold increase in code output at negligible cost, and a more important effect: work deferred for years as too difficult has become worth attempting. It describes the techniques that made this possible, and asks whether we are all now redundant.

N.J. Nelson B.Sc. C.Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

---

1. Introduction

At last year's conference I complained that Forth was stuck in a rut, and proposed a number of radical changes to get it out. This year I want to look at a rut of a rather different kind — one that the rest of the programming world has climbed out of at considerable speed, while we have been standing at the bottom of it debating the semantics of `WITHIN`.

Every mainstream language now has a competent AI assistant available to it. Ask one to write a Python decorator, or a React component, or a chunk of Rust, and you will get something that compiles and usually works. The obvious question for us is whether any of this is of the slightest use to a Forth programmer, and if so, how much.

The short answer is yes, considerably more than I expected — but the gains arrive in quite different places from the ones I had assumed.

2. Timeline of AI development

AI is a great deal older than Forth, and this is not its first boom.

- **1950** — Turing proposes his imitation game.
- **1956** — The Dartmouth workshop coins the term "artificial intelligence", and predicts that most of the problem will be solved over one summer by ten men. It was not.
- **1958** — McCarthy invents Lisp. For the next thirty years, AI and Lisp are effectively synonymous.
- **1966** — ELIZA. A few hundred lines of pattern matching convinces a surprising number of people that it understands them. This lesson has had to be relearned every fifteen years or so since.
- **1969** — Minsky and Papert publish *Perceptrons*, and neural networks go into hibernation for a decade and a half.

- **1970s** — Chuck Moore invents Forth. Meanwhile the first AI winter sets in.
- **1980s** — Expert systems, inference engines, Prolog, and the Japanese Fifth Generation project. Lisp machines sell at eye-watering prices. This is also the era of Forth's commercial peak, and the two worlds occasionally meet: both communities were interested in small interactive systems that the user could extend at run time.
- **1987–1993** — The second AI winter. The Lisp machine market collapses, and "artificial intelligence" becomes a phrase you avoid in a funding application.
- **1997** — Deep Blue beats Kasparov, by brute force rather than insight.
- **2012** — AlexNet wins ImageNet by a wide margin. The approach written off in 1969 turns out to have been merely short of computing power and data.
- **2017** — The transformer architecture is published. This is the one that matters for us.
- **2020–2022** — GPT-3, and then in November 2022 ChatGPT, which puts a usable language model in front of the general public.
- **2023–2024** — Models become useful at coding rather than merely amusing. Assistants move into the editor.
- **2025–2026** — The agentic era. The model no longer suggests a line of code; it reads your source tree, edits several files, runs the compiler, reads the errors and tries again. This is the point at which the tools became interesting to me.

Two previous winters should make us cautious. But both earlier booms collapsed because the technology did not work, whereas this one has produced something that demonstrably does, and that people are willing to pay for. That is a material difference.

3. Personal experiences of early AI

I have been buying artificial intelligence, in one form or another, for about thirty years. Almost all of it disappointed me, and the pattern of *which* parts disappointed is the most useful thing I know about the subject. I should also say, because it matters for section 9.1, that I began my career writing assembly language, and can still remember what it felt like to move to a high level language.

Speech to text. An obvious productivity improvement: I can talk faster than I can type. It was expensive. You had to train it to recognise your own voice, which took an evening of reading passages aloud to a computer, feeling foolish. Having done so, it did not work. The correction time exceeded the typing time it was supposed to save.

Machine translation. We sell into a number of countries and our applications are multilingual, so this looked like a real saving. Again expensive, and again the output needed so much tidying by somebody who spoke the language that it was not worth having.

My first satnav, a standalone box on the windscreen. This one was *brilliant*. It knew every rat run in the country, including several I had lived near for years without noticing. It also, on one memorable occasion, routed me confidently into the back of a five mile queue around the very next corner. It knew everything about the road network and nothing whatever about what was happening on it.

Voice control in my car, about seven years ago. It argued with me.

An Android phone, at about the same time. It understood me perfectly, first time, without any training at all.

A VW, two years ago. My grandchildren were five, and had substantially larger vocabularies. I do not say that for comic effect: a system that makes you repeat yourself three times while driving is not merely annoying, it is dangerous.

A new Skoda, this year. Better. The vocabulary is still tiny.

The moral is not that AI does not work. Look at where the line falls. Everything that failed was bespoke — a system I had to train myself, or one trained on a small corpus, or one frozen at the moment it left the factory. Everything that succeeded was trained on an enormous body of data collected from everybody, and kept up to date.

The phone understood me because millions of people had already talked to it. The satnav knew the rat runs because somebody had surveyed every road in Britain, and it missed the queue because in 2003 nobody was collecting that. The car does not understand me because a car is shipped once and then abandoned, running a small model on a small processor with a vocabulary chosen in Wolfsburg some years before I bought it.

Hold that thought until section 5, because it is exactly the situation Forth is in.

4. Starting to code using AI — Google Gemini

My first attempts, using Google's Gemini, were unstructured. I would open a browser window, describe what I wanted, and paste the result into the editor. The experience was mixed.

What worked immediately was everything that was *not* Forth. Our main application does not exist in isolation. It talks to a database, it talks to GTK3, it talks to a dozen C libraries, and it produces output in several human languages. For all of these the assistant was better than a search engine and very much faster than the documentation:

- **SQL.** Complex queries against our schema, written correctly first time. Our dashboards are configured rather than programmed, and the configuration is largely SQL, so this alone was worth the money.
- **The GTK3 API.** I asked how to make `GtkCalendar` display in another language. It took ten seconds to tell me that you cannot. I had previously spent a day trying to find out how. We wrote our own, which is better.
- **Regular expressions.** No further comment is necessary.
- **Translations.** We hold phrases in a database in a number of languages. A model produces a first pass at a new language in seconds, which a native speaker then corrects. Correcting is much faster than translating.
- **Boilerplate in other languages.** Utility scripts, build glue, one-off data conversions.

What surprised me was Forth. I had assumed it would be hopeless. It was not. It wrote Forth, it knew what VFX was, and the code ran.

It was, however, unmistakeably somebody else's code.

5. Does it actually speak Forth?

This is the question in the title, so it deserves a section of its own. I went in expecting to answer "no". The honest answer is "yes — but with a pronounced accent, and one I recognised".

a) It handles the stack better than I do. I report this first because it is the opposite of what I expected, and of what everyone I have described it to expects. The universal assumption is that a system generating one token at a time cannot keep track of stack depth across thirty words, and that Forth is therefore uniquely resistant. That assumption is wrong. It tracks the stack accurately, it produces correct stack comments, and on more than one occasion it has found a depth error in *my* code that I had read past several times.

On reflection I should not have been surprised. We have spent forty years writing `( in --- out )` on every definition. Our published corpus, small as it is, is annotated throughout with machine-readable type information, by a culture that treats omitting it as bad manners. No other language has been so consistent about this. We prepared our source code for exactly this without intending to.

b) The corpus is small at training time and surprisingly large at working time. A model learns to program by reading public source. There is perhaps five or six orders of magnitude more public Python than public Forth, and much of the Forth that exists is forty-year-old listings, toy implementations written in something else, or blog posts explaining what the stack is.

That describes what the model brings to the table, not what it has to work with. Once started, it has a great deal of code available: our own application, the supplied example sources, and — the item I had not thought of, which changes everything — the source of the compiler itself. VFX is not a black box. The model can read how the system it is writing for is actually built.

The obvious prediction is that it therefore writes Forth badly. It does not. What it does is stranger. The Forth it wrote for me was styled exactly like Stephen Pelc's: the same spacing conventions, the same naming habits, the same instinct about where to break a definition. I could have picked it out of a line-up.

There is nothing wrong with his style. In Forth we all have our own, and most of us can recognise each other's on sight. His is not bad, just different from mine.

That deserves more attention than it gets, because it is close to unique to us. In most languages house style is imposed by a formatter and personal style is squeezed out; you cannot identify the author of a block of Python from the shape of it. Forth is free format, so individual authorship stays visible. Every one of us writes a recognisable hand — which is precisely why the effect showed up so plainly. Give a model Forth to imitate and it will imitate a *person*, because in Forth there is a person there to imitate.

There are two possible explanations, and I want to set out both, because I think the second is correct and it is the more interesting.

The first is a claim about training: in a corpus this small there is no average to regress towards, so one prolific published author stops being a data point and becomes the distribution. In a large language you get the mean of a million programmers, which is why machine-written Python is so characterless. In Forth there is no mean.

The second is simpler. It wrote in his style because I had given it his source code to read. The compiler sources and the supplied examples are his work, and I had pointed the model straight at them. It was doing what it had been asked to do — match the surrounding style — and the surrounding style was not mine.

I now believe the second explanation accounts for essentially all of it, and section 5.1 describes the experiment that settled the matter.

c) It looks up the standard, and the standard is decades out of date. This was the one real practical problem, and I must be careful here, because the point is easy to state rudely and I do not mean it that way.

Asked to write Forth, Gemini did the responsible thing: it went and looked up the specification. It found the ANS document and wrote accordingly. When I first drafted this section I put that down to a search engine surfacing an old copy, and told myself the lesson was about discoverability. On reflection that is not the lesson at all, because it would have made very little difference had it found the current document instead.

Forth 2012 is itself fourteen years old, and is in substance a careful refinement of a specification written in 1994. Whichever document the model consults, what it learns is Forth as it stood more than thirty years ago: no strings worth the name, no threads, no way to call an external library, and a set of structure words that I have never seen anybody use in earnest. The model was not misled by finding the wrong document. It was accurately informed by a document describing a language that none of us has written for decades.

I made this argument last year and will not repeat it at length. What is new is the consequence. The gap between the standard and the language as practised used to cost us new programmers, which was bad enough and easy to ignore. It now also shapes what the machines write, silently and by default, because consulting the specification is exactly what a well-behaved assistant ought to do.

The practical remedy takes one line in the agent directory: tell it not to rely on the standard, and tell it what to use instead. It complies without argument. But I would rather not have to write that line, and I do not think we should be comfortable that it is necessary. None of this is a criticism of the people who maintain the document, who have done careful work within the remit they were given. It is an observation about the remit.

d) Our own dialect does not exist anywhere. Almost nothing in our main application is standard Forth, or MPE's Forth either. `Z"`, `Z+`, `ZFORMAT`, `VINDEX`, `VMATRIX`, `STRINDEX`, `ENUM<<`, `SIG:`, the whole of FQL — none of it appears in any training corpus, because all of it lives on our disks. Neither do our house rules: that `WITHIN` and `CASE` are redefined at the top of every build file, that strings are zero terminated, that `VALUE` is preferred to `VARIABLE`, that new code is 1-based.

So the difficulty was never capability. The code arrived fluent, correct, and written to somebody else's standards — which in a codebase we intend to maintain for another twenty years is not a small objection. Consistency of style is not vanity. It is what lets you read a file you last touched in 2011 and understand it before lunch.

This is my satnav again. It knows every road in the country, but not the local conditions, and it has no way of knowing them unless somebody tells it.

5.1 The single most important thing in this paper

If you take nothing else away, take this: **the agent directory is where the work is.**

Modern assistants read a set of markdown files placed in a directory alongside your project. Everything above — the out-of-date standard, the borrowed style, the unknown house words — is fixed in those files, and only in those files. They are not documentation and they are not configuration. They are the most important source code in the project, because they determine the character of everything the machine subsequently writes.

Ours tells it, among much else:

- not to rely on the published standard, and what to use in its place;
- that `WITHIN` and `CASE` are redefined at the top of every build file, and why;
- that strings are zero terminated, and that `Z"` is how you make one;
- that `VALUE` is preferred to `VARIABLE`, with `->` for assignment;
- that loop indices are 1-based in new code;
- the stack comment convention `( in --- out )`, always present;
- that trailing `\` comments are aligned.

As soon as I understood this, and wrote our company coding rules down properly, the behaviour changed completely. **It stopped writing Pelc-Forth and started writing Micross-Forth.** It has stuck to the rules ever since, cheerfully and without argument, which is more than I can say for some human programmers I have employed.

That also settles the question left open in section 5(b). If the style had been a property of the training data, writing a few markdown files could not have shifted it. It shifted immediately. The style comes from what is in front of the model, not from what is behind it — which means it is under your control, provided you take the trouble to say what you want.

One further practice is worth stating: **give it the source, and be deliberate about whose.** Point the assistant at our own code and ask for a new word in the style of the surrounding file, and you get it. Point it additionally at the compiler sources when you need to know what a system word actually does — a luxury no C++ programmer has ever had. But choose consciously, because whatever you show it is what you will get back.

Forth's brevity is a real advantage here. Our main application is approximately 80,000 lines in 124 source files, an average of some 645 lines per file, which is about what a house style favouring short definitions and one subject per file would produce. Eighty thousand lines of Forth is of the same order as what a current model can take in at one time, so the relevant subsystem, the agent directory rules and the compiler sources for the words in use all fit together comfortably.

A word on comparisons, since somebody will raise it. The published language-level tables — Capers Jones's is the one usually cited — put Forth at 64 source statements per function point against 128 for C, and 53 for C++ and Java. Taken at face value that says Forth is merely twice as dense as C, and slightly *less* dense than C++. That is not my experience. My experience is that C++ is very much longer, and that the only language I would call comparable is Python with the comments taken out.

I think those figures are measuring 1996 Forth. The tables date from that era, Forth was sized by comparison against COBOL rather than counted directly, and every Forth dialect in the list carries an identical level — which suggests one family estimate rather than a measurement. They also cannot see metaprogramming. A single line of ours can define a whole family of words; `ENUM<<`, `SIG:`, `CallProc:` and the whole of FQL exist to do precisely that, and backfiring has no way to count what it cannot see. I offer this as experience rather than proof, and I would be glad to be measured properly.

For this paper the ratio matters less than the absolute size. Whatever the multiplier, the subsystem I am working on, its house rules and the compiler sources for the words in use all fit in front of the machine together — and that is a claim I can demonstrate on a laptop rather than argue about.

Our agent directory is **426 lines in 14 markdown files** — roughly one line of English governing every 188 lines of Forth, and line for line the most valuable text in the repository.

6. Moving to Anthropic — Opus, then Fable

Gemini wrote pretty good Forth. It needed a lot of guidance to do it — steering, correcting, reminding, and a fair amount of rewriting to house style afterwards. Even so, it roughly doubled the rate at which I could get through the work I had been given.

I should be honest about what I did with that. I took it as shorter days. The issues I had been assigned got finished and I simply stopped work earlier — I have a home office, so there is no commute to mark the end of the day, just the moment when the list is clear. This matters because it means the "before AI" baseline in section 8 is not quite AI-free, and yet the committed-lines figures barely move. The gain went into my afternoons rather than into the repository. What changed in May 2026 was not only the capability of the tools but what I chose to do with the time they released.

Claude Opus 4.6 wrote noticeably better code. Less guidance, fewer corrections, and a much better instinct for what I had actually asked for rather than what I had literally typed. This is where the bulk of the gain reported in section 8 came from, and I say so plainly because the temptation is always to credit the newest thing on the shelf. The measured step change begins on 1 May 2026, and Fable was not available until 9 June.

Claude Fable 5 writes almost faultless Forth. The improvement over Opus is real but incremental, and it is not what produced the numbers. What it changes is what happens to the code before I ever see it: **it compiles and tests the code itself, and only then shows me the result.** I am no longer reviewing a suggestion. I am reviewing something that has already been round the loop I would have put it through, with the failures found and fixed on the way.

That is the point at which the tool stops being an assistant and becomes something closer to a colleague. Everything in section 3 that failed, failed because the correction burden exceeded the saving. Dictation, translation, the car: in each case the machine produced a draft and left the tidying to me, and the tidying was the expensive part. A system that does its own tidying is a different category of thing.

It is also, I suspect, why Forth suits this so well. Our edit-compile-execute-test cycle is extraordinarily quick, because everything except the edit happens inside Forth. That has always been our advantage over the C++ world, and it is a bigger advantage when it is the machine going round the loop rather than me: it can iterate a dozen times in the time a C++ build would take to fail once.

Two practical matters worth recording:

- **Availability.** Fable 5 was released on 9 June 2026, withdrawn three days later to comply with US Department of Commerce export controls, and restored on 1 July when those controls were lifted. Anyone thinking of building a commercial dependency on a frontier model should note that three-week gap. *(Anthropic's statement: https://www.anthropic.com/news/fable-mythos-access)*

- **Routing.** Fable ships with safeguards that divert some queries to Opus instead. Anthropic state this is tuned conservatively and fires in fewer than 5% of sessions. In practice you occasionally get an answer from a different model without being told which. For those of us writing industrial control software, that is the kind of thing one likes to know.

7. You get what you pay for

The free tiers are, for our purposes, a waste of time. They are the demonstration model in the showroom: smaller models, which forget the house rules halfway through and cannot see your source tree.

This has a predictable and depressing effect on the wider conversation. A great many people have formed a firm view of what AI can do for programmers on the basis of ten minutes with the free version — in much the same way that a great many people have formed a firm view of Forth on the basis of ten minutes with the standards website. In both cases the verdict is "quaint, but not for serious work", and in both cases it is wrong.

The paid tiers cost between £100 and £200 per month. Analysis of our git history, set out in the next section, shows that before AI each line of finished, issue-tracked code in our Tracknet repository cost £3.45 in development time. A £150 subscription therefore pays for itself at **44 additional lines a month**. That is one modest afternoon.

I am not aware of any other purchase available to a software company where the break-even point is so low that calculating it feels faintly absurd. If your program runs too slowly, I have said before, spend five Euros on a faster PC, because it is cheaper than a programmer's time. The same reasoning applies here with about three orders of magnitude more force.

There is a second and less obvious cost: checking. Everything the model produces must be read. Forth's brevity helps enormously — reading a screenful of Forth takes a fraction of the time of reading the equivalent in a verbose language, and our rule that every definition carries a stack comment and aligned trailing comments means wrong code tends to look wrong. Forth's much-derided terseness is a competitive advantage in the age of machine-generated code, because the review burden scales with the volume of text a human has to read, and we produce less of it per unit of work than anybody.

8. Productivity and cost effectiveness

These figures are not estimates. They come from the complete git history of our Tracknet repository — the 80,000-line main application described in section 5.1, comprising 1,183 commits between April 2020 and 23 July 2026 — counting only hand-written source files (Forth, SQL, CSS, Python) and excluding generated UI files, configuration, binary data, bulk imports and reformats.

Measure	Before AI (Jan 2024 – Apr 2026)	With AI (1 May – 23 July 2026)
Source lines added	21,664 over 28 months	22,594 over 12 weeks
Coding days	179	32
Lines per coding day	**121**	**706**
Commits per month	about 11	about 34

In twelve weeks the repository gained more code than in the preceding two and a third years combined. The rate per coding day rose by a factor of **5.8**. To put it another way, an application built up over six years grew by more than a quarter in three months. July 2026, still unfinished when counted, was already the most productive month in the repository's history apart from the initial import. Commit frequency tripled, and every commit in the AI period remains tied to a tracked issue, exactly as before.

The step change is not subtle, and is best seen in the monthly figures:

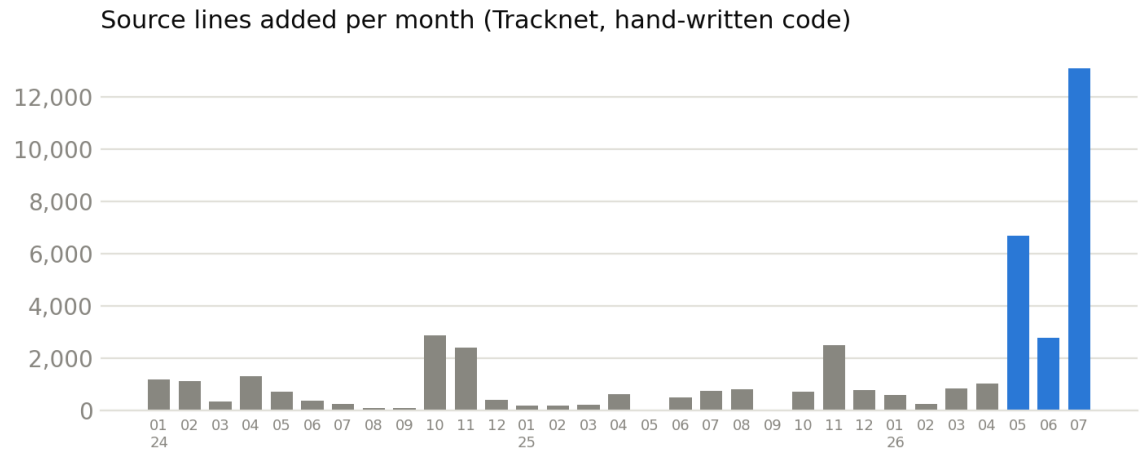


Figure 1. Hand-written source lines added per month to the Tracknet repository, January 2024 to July 2026. Generated UI files, configuration, binary data, bulk imports and reformats are excluded.

Lines are only half the story, and the less interesting half. Counted by tracked issues closed:

Measure	Before AI	With AI
Git issues completed	**about 7 per month**	**49 in three months, about 16 per month**

That is a 2.3-fold increase in issues against a 5.8-fold increase in lines. The two figures do not agree, and the disagreement is the most interesting result in this paper. I return to it in section 8.1.

A note on the money. The report values the extra output at £3.45 per line, derived from a £60,000 salary benchmark worked at twenty hours a week. This is a *market rate for equivalent developer time*, not my own remuneration, which is not on that basis at all. Read properly, the figure says what it would have cost to buy 20,473 lines of finished, issue-tracked code at prevailing rates: some £70,560 in twelve weeks, or around £309,000 annualised if the pace holds. The subscription costs between £100 and £200 per month. I have checked that ratio several times, because it does not look credible written down.

Now the caveats, which I would rather put in myself than have supplied for me.

Lines of code is a poor metric, and a particularly poor one in Forth. I am uncomfortably aware that last year I stood at this conference and boasted about *removing* twelve thousand characters from our source. Our whole culture holds that the best code is the code you did not have to write. A Forth programmer measuring output in lines per day is like a surgeon measuring output in incisions per day. The figure should be read as throughput, not value.

I am working longer hours than I was. I am contracted for about twenty hours a week and have been exceeding that lately, for the straightforward reason that the work has suddenly become interesting again — itself a result worth reporting, though not one that appears in any table. It means "lines per coding day" is measuring slightly longer days at the AI end, and some part of the 5.8 is hours rather than rate. The issues figure carries the same qualification.

The baseline is not quite AI-free. I had been using Gemini for some time before May 2026, but took the gain as shorter days rather than more output, so it barely shows in the committed lines. The step change in May is real: it is the point at which the gain started going into the repository instead of into my afternoons.

AI-assisted work involves more churn. Deletions have risen too. Excluding generated and configuration files limits this but does not eliminate it.

Most of this is Opus, not Fable. The measured period begins on 1 May 2026; Fable was not released until 9 June and was then unavailable for three weeks. The newest product tends to collect credit for gains it did not produce.

It is not a controlled experiment. It is the same developer on the same product, which usefully controls for skill and domain, but the feature mix varies month to month and July was incomplete when counted.

Quality is not directly measured. What can be said is that the defect-fix pattern in the log is unchanged, and every commit remains tied to a tracked issue.

And the report was prepared by the tool being measured. I record this plainly because somebody will otherwise record it for me. The underlying git history is objective and independently checkable, and I have satisfied myself that the counts are right — but the reader is entitled to know that the productivity analysis of Claude Code was carried out using Claude Code.

Even discounted for all of that, the gain is real, and unevenly distributed. It is enormous for the large part of our work that is not Forth at all: database, user interface toolkit, libraries, translations, documentation, test data. It is substantial for the Forth itself, but only once the house rules are written down. And it is close to zero for what actually consumes the most time in a Forth shop, which is deciding what the program ought to do.

8.1 The backlog — the result that matters

Every long-lived project has a category of issue that never gets done. Not because it is unimportant, and not because nobody has noticed, but because every time it reaches the top of the list somebody estimates it, winces, and puts it back. Too difficult. Too time-consuming. Too much risk of breaking something that works. Ours had items that had been sitting there for years.

I have been working through them.

That, rather than the lines per day, is what I would want this conference to take away. The productivity figures describe doing the same work faster. This is different in kind: work that was never going to be done at all is being done. No improvement in my typing speed would ever have cleared that backlog, because the obstacle was never typing. It was that the expected cost of the attempt exceeded what the item was worth, so the calculation came out negative every time it was made, for years. Lower the cost of the attempt by a factor of five and it comes out differently.

This also explains the discrepancy between the two measurements above. Issues closed rose 2.3-fold; lines written rose 5.8-fold. The ratio of lines to issues has roughly doubled, which is what you would expect if the issues now being tackled are the big awkward ones that had been deferred. I am not writing more code per issue because the machine pads. I am writing more code per issue because I am finally attempting the issues that needed a lot of code.

There is a lesson here for anyone put off by arguments about metrics. The question is not how much faster it types. The question is what is now worth attempting that was not worth attempting before — and for a codebase with twenty years of deferred work in it, that answer can be very large.

9. The future — are we all redundant?

The honest answer is: not yet, and probably not in the way the newspapers suggest.

The pessimistic case is straightforward. Model capability has improved faster over the last three years than most of us expected, the agentic tools now close the loop without a human in it, and there is no obvious reason for the curve to flatten just because it would suit us if it did.

I began this paper intending to offer three reasons why Forth might prove more resistant than most. Two did not survive the writing of it, and I set them out as they stood, because the way they failed is more instructive than a tidy list of one.

The first was that Forth's obscurity is a moat. A model cannot learn a corpus that does not exist, and every serious Forth application is written in a private dialect living on one company's disks. Section 5 disposes of it. Our private dialect was absorbed not over years of training but over an afternoon of writing markdown files. Obscurity is no defence when the thing you are hiding behind fits in a directory.

The second was that the stack is genuinely hard for these systems. This is the defence we would all have reached for first: that concatenative code with implicit state is the worst case for a next-token predictor. It is not a defence either. It handles the stack better than I do, and section 5 argues that our own commenting discipline is part of the reason. We spent forty years annotating our source with machine-readable type information, and should not now be astonished that a machine can read it.

The third was that specification is the bottleneck, not typing. This one stands. Nobody has automated the part of the job where you go to a factory in Herefordshire, watch what the machine actually does, and work out what the customer needed rather than what they asked for. I would not care to guess how long that remains true, but it is true today.

So of the three protections I expected Forth to have, two were imaginary and the third is not specific to Forth at all.

There is a further uncomfortable observation. Suppose AI makes mainstream languages three times more productive and Forth 5.8 times. That sounds like good news, and for me personally it is. But our commercial argument has always been that a small Forth team can out-produce a large team using something else. If everybody's productivity multiplies, the *ratio* is what matters, and the ratio is now set by tooling rather than by language design. That is a more serious threat to Forth's commercial case than anything currently being debated in the standards process.

Which brings me back to last year's paper. If Forth is to survive the next decade it needs data types, proper string handling, threads, a way to call external libraries, and a specification describing what real applications actually need. I argued all of that a year ago on the grounds that it would help attract programmers. The argument has since acquired a second and quite unexpected constituency, as section 5(c) describes: the machines those programmers will work alongside read the specification literally, believe it, and write accordingly. We are no longer only failing to describe the language we use. We are actively teaching it wrong.

9.1 What this actually feels like

I have been trying for some months to describe the sensation of working this way, and I think I have placed it. It is not new. I have felt it once before.

I started out writing assembly language, and I can clearly remember moving to a high level language for the first time: the discomfort of it, the suspicion that something was being taken away from me, and then the gradual realisation that I had been moved up a level and could see further.

This is the same feeling, and the parallel is exact rather than poetic.

I used to write very carefully composed Forth, and then, when it mattered, go and look at what the VFX optimiser had made of it. I did not write the machine code. I wrote the thing above the machine code, and inspected the output to satisfy myself it was sound.

I now write very carefully composed instructions, and then go and look at what Fable has made of them. I do not write the Forth. I write the thing above the Forth, and inspect the output to satisfy myself it is sound.

It is coding, at the next level up.

That is the honest answer to this section's title. We are not redundant. We have been promoted, in the same way and for the same reason that assembly programmers were promoted forty years ago — and the skill that transfers is precisely the one this community already has in abundance: composing something carefully, then checking what the layer below actually did with it. Forth programmers have always looked at the generated code. We are, oddly, better prepared for this than most.

I will not pretend the transition was painless the first time. Not everyone made it, and the number of people needed for a given job did change. But the work did not disappear; it moved. And speaking for myself, at this stage of a long career, the surprise has been how much I am enjoying it. That is why I have been exceeding my twenty hours, and it is the one thing in this paper I did not anticipate when I started.

10. Conclusion

Does AI speak Forth? Yes, and better than I expected — including the stack, where I was most confident it would fail. What it cannot guess is *our* Forth, and telling it is the whole of the practical work: 426 lines of house rules in the agent directory, which are now the most important text in the repository.

The measured result is 121 to 706 lines per coding day, for rather less than the cost of a working afternoon per month. But the number I would want remembered is not that one. I have spent this summer closing issues that had sat in the tracker for years, marked too difficult to attempt. The tools did not make me a faster typist. They changed which jobs were worth starting.

And if you want to know what it feels like, it feels exactly like the day I stopped writing assembly language.

11. Did AI write this paper?

Yes, substantially. It would be absurd to pretend otherwise in a paper on this subject.

I supplied the experiences, the opinions, the git figures and the judgement about what mattered. The machine read my previous EuroForth papers to establish the style, structured the argument, drafted the prose, and extracted Figure 1 from the productivity report. I then corrected it, repeatedly — and the corrections were of exactly the kind described in section 5: fluent, confident, plausible, and in several places quite wrong.

EuroForth 2026

Autotranslation within a Forth application — two different approaches

Abstract

Every piece of user-visible text in our main application is a “phrase”, held in a database with one row per language. Translations used to be supplied by humans, eventually. This paper describes making them automatic, twice: first through Google Translate over a raw socket, then through Claude via a relay server. The first works, and is free, and its translations are mediocre for an interesting reason: there is nowhere to put the context. The second approach exists because of that limitation, and is now in service.

N.J. Nelson B.Sc. C.Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

---

1. The phrase system

Tracknet, our main application, contains no user-visible English strings. It contains *phrases*, declared in source with `P"` or in the “Glade” user interface designer, and stored in a database table with one row per language per phrase. The phrase system in its current form was described in my 2019 paper to this conference [1], which introduced the `P"` approach; its roots go back to my multilingual application papers of the 1990s [2]. About eight languages are currently in service at customer sites; the ISO-code mapping table is ready for 38.

`ADDPHRASE` and its siblings insert the English base row and create a row for each configured language. Until the work described here, the translated rows were filled in by human translators, eventually. A new phrase fell back to English everywhere until then. The aim of this work: have a machine fill them in at the moment the phrase is created.

One complication runs through everything that follows. A diagnostic phrase may contain the substitution codes `#`, `#I`, `#C` or `#Q`, replaced at display time by a number, an item, a count or a quantity. These must pass through translation verbatim, in the right position, or the diagnostic becomes nonsense.

2. First approach: Google Translate

2.1 Over a raw socket

Google's free `gtx` endpoint takes an HTTP GET and returns JSON. `Autotranslate.fth` (about 300 lines) speaks to it directly over a TCP socket — no HTTP library, no JSON library. `TRANSLATE-PHRASE` URL-encodes the phrase, resolves, connects, sends, receives, and parses the translation out of the reply. On any failure the language row falls back to the English text.

The request that goes down the socket is composed as one chain. `ZCAT` appends a Zstring to a destination and leaves the destination address, so the whole HTTP request is a single run of `ZCAT`s with one `DROP` at the end — no length arithmetic, no running end pointer:

```
: FORMAT-HTTP-GET { pcode pescaped pdest -- } \ Format HTTP GET request
  0 pdest C!                                              \ Start with an empty string
  pdest                                                   \ Destination for the chain
  Z" GET /translate_a/single?client=gtx&sl=en&tl=" ZCAT \ Request line: path prefix
  pcode                                           ZCAT \ Language code
  Z" &dt=t&q="                                    ZCAT \ Query parameter
  pescaped                                        ZCAT \ Escaped query
  Z"  HTTP/1.1"                                   ZCAT \ HTTP version
  ZCRLF                                           ZCAT \ End of request line
  Z" Host: translate.googleapis.com"              ZCAT \ Host header
  ZCRLF                                           ZCAT \ End of host header
  Z" Connection: close"                           ZCAT \ Connection close header
  ZCRLF                                           ZCAT \ End of connection header
  ZCRLF                                           ZCAT \ Empty line ends the request
  DROP                                                 \ Discard destination pointer
;
```

`Connection: close` does double duty: the server's end-of-reply is the end of the connection, so the receive loop needs no content-length handling — read until the far end closes.

The endpoint is the free, undocumented interface used by Google's own web translator; everything we know about it was learned by capturing live responses. Google's documented product is the commercial Cloud Translation API [3], which requires an API key and billing. We considered it and decided against, for two reasons: at our volume it would cost about $1,000 a year — not onerous, but a consideration against free — and its documentation, when we tried to follow it, kept ending in dead links. The undocumented endpoint at least makes no promises. Building on an undocumented endpoint is a stated risk in section 4 — it may change or vanish without notice, and when it broke our parser (section 2.3), there was no specification to consult.

Connecting is the part that needs care. A plain `connect` on a machine with no internet can block for minutes, which would hang every compile that adds a phrase. So the socket is put into non-blocking mode, the connect is initiated, and a `select` with a one-second timeout waits for it to complete:

```
: CONNECT-TIMEOUT { psock paddr | pstatus -- f }
  psock SETNONBLOCK                         \ Non-blocking mode
  psock paddr SOCKADDR_IN connect -> pstatus
  pstatus 0= IF                             \ Connected at once
    psock SETBLOCK  TRUE
  ELSE
    errno @ EINPROGRESS = IF                \ Still connecting
      WRITEFDS FD_ZERO
      psock WRITEFDS FD_SET                 \ Watch this socket
      1 SO_RCVTIMEO_VAL tv_sec !            \ One second limit
      0 SO_RCVTIMEO_VAL tv_nsec !
      psock 1+ NULL WRITEFDS NULL
      SO_RCVTIMEO_VAL select -> pstatus     \ Wait, write-ready
      pstatus 0> IF
        psock SETBLOCK  TRUE                \ Connected in time
      ELSE
        FALSE                               \ Timed out
      THEN
    ELSE
      FALSE                                 \ Refused outright
    THEN
  THEN
;
```

(`SETNONBLOCK` and `SETBLOCK` toggle `O_NONBLOCK` with `fcntl`.) After that, sending the request is one line: `psock preq preq ZCOUNT NIP WRITEN`.

2.2 The context problem

Ask a translation engine for “washer” and you may get the machine or the flat metal ring. In a commercial laundry application this matters, and the `gtx` endpoint offers no field for saying which domain you are in. It takes a source language, a target language, and text.

The only place to put context is therefore in the text itself:

```
: PREPEND-CONTEXT { pz$phrase pdest -- }  \ Prepend the context
  Z" Commercial laundry: " pdest ZMOVE    \ Copy the prefix
  pdest pz$phrase ZCAT DROP               \ Append the phrase
;
```

The translated prefix is stripped back off the reply — everything up to the first colon. It works, mostly, and “mostly” is the reason approach two exists:

- The context is one noun phrase. We cannot say that “RUN” is a button and needs the imperative, that this phrase must fit a 12-character display, or that “bag” must be translated consistently with two hundred earlier phrases.
- Stripping “up to the first colon” assumes the colon survives translation in all languages.
- The engine translates our substitution codes. `#I` is not English, but the engine does not know that. `PROTECT-HASH` shields the codes before sending and `RESTORE-HASH` puts them back — the same hack again, structure smuggled through a channel that accepts only prose.

None of these is a defect in the engine. They are all one defect in the interface: everything we know that is not text must be disguised as text, and undisguised on the way back.

2.3 The silent truncation bug

Adding the sequence on/off switch tooltip:

> “Switches this sequence on or off. When off, the sequence does not run.”

the French, German and Spanish rows contained only the first sentence. No error, no fallback — the second sentence silently gone, in every translated row.

The endpoint (undocumented, so this was found by capturing live responses) segments the query at sentence boundaries and returns one `["translated","original",...]` sub-array per sentence. Our parser copied up to the first unescaped double quote — the first segment only. Single-sentence phrases have one segment, so the bug bit only phrases containing a full stop, and survived six weeks.

The fix loops over segments, splitting at the four-character separator:

```
\ Separator ],[" between sentence sub-arrays in the response
\ JSON. Unambiguous because a double quote inside a JSON string
\ is always escaped, so ],[" cannot occur within text.
CREATE ZSEGSEP CHAR ] C, COMMA C, CHAR [ C, QQ C, 0 C,
```

(Built with `CREATE ... C,` because a `Z"` literal cannot contain a quote character.) A `ZLIMIT` was added before the final copy: callers pass 251-byte locals sized to `MAXPHRASELEN`, and a multi-sentence translation can legitimately run longer than its English source.

The fix was verified with a standalone harness running the parser verbatim over the captured responses (one, two and three sentences, plus garbage input), a clean fresh-session compile, and live calls returning both sentences in French, Italian and Spanish. The captured responses and harness are kept under version control with the fix, so the evidence outlives it.

Making the file loadable standalone flushed out two collateral defects, one a real bug in an earlier kernel patch: a stray cell left on the stack, and a 64-bit store writing four bytes past the end of a 5-byte JMP. The method is the Forth story — bisecting an include chain by stack depth takes minutes.

2.4 The background queue

Compile-time translation can block for a second per language. Runtime phrase additions — a button text typed into the GUI — cannot, and must not fail on a machine with no internet. So runtime additions create every language row immediately with the English text, and queue one job per (table, id, language):

```
32 CONSTANT MAXTRANSJOBS               \ Slots in the queue
3  CONSTANT TRANSTRIES                 \ Tries before giving up
```

The queue is a circular buffer with no locks: `TRANSIN` is written only by the queuing thread and `TRANSOUT` only by the background task — single producer, single consumer, so the indices never race. `TRANSTASK` drains it, writing each success to the database row and the live in-memory phrase table, so the GUI updates without a restart. Failures retry three times at 60-second intervals, then leave the row in English — where it would have been anyway.

At this point the machinery was sound. The translations were the remaining problem, and the limitation was in what the wire protocol could carry.

3. Second approach: Claude via a relay

3.1 Context becomes a first-class citizen

A large language model inverts the situation. Context is not squeezed into the text; it is the primary input. The instruction can say, in plain English, at whatever length is useful: this is a commercial laundry application; these are button labels, so use the imperative; anything matching `#`, `#I`, `#C`, `#Q` passes through untouched; return only the translation.

Better still, it can *show* rather than tell. The request includes examples of similar phrases that a human has already translated into the target language, drawn from the phrase database itself. Twenty years of native-speaker translations stop being merely the answers we already have and become the specification for the answers we do not: terminology, register, and length habits are carried by example, which is exactly how one would brief a human translator.

The examples are gathered by the client at translation time. `MATCH-WORDS` extracts the phrase's significant words, a score expression is composed from them, and one query ranks every phrase in the database — GUI, diagnostic, sequence and IO tables together — by resemblance to the phrase in hand. The best few, with their approved translations, are appended to the request as JSON pairs:

```
\ Fill XJ[ with the request tail: up to MAX-EXAMPLES approved
\ translations resembling zphrase
: EXAMPLES-JSON { zphrase zlang | pd pn pcount pwords[ /MATCHWORDS ] pexpr[ /SCORE ] -- }
  XJ[ QQ +C -> pd                                  \ Tail closes the target string
  0 -> pn                                          \ No examples yet
  zphrase pwords[ MATCH-WORDS -> pcount            \ The phrase's significant words
  pcount IF                                        \ There are words to match
    pwords[ pcount pexpr[ SCORE-EXPR               \ The score they define
    SQL| SELECT DISTINCT en, tr, score FROM ( |
      zlang pexpr[ zphrase Z" phrases"     +EXAMPLE-ARM  |  UNION ALL |
      zlang pexpr[ zphrase Z" diagphrases" +EXAMPLE-ARM  |  UNION ALL |
      zlang pexpr[ zphrase Z" seqphrases"  +EXAMPLE-ARM  |  UNION ALL |
      zlang pexpr[ zphrase Z" IOphrases"   +EXAMPLE-ARM
      |  ) x WHERE score > 0 ORDER BY score DESC, LENGTH(en) LIMIT | MAX-EXAMPLES FQL-N+
    |SQDO                                          \ For each example found
      pn 0= IF                                     \ First example opens the array
        pd JEXH >Z -> pd
      ELSE                                         \ Later examples follow a comma
        pd [CHAR] , +C -> pd
      THEN
      pd JEX1 >Z -> pd                             \ {"en":"
      0 PCOL ESC[ JSON-ESCAPE  pd ESC[ >Z -> pd    \ Base phrase, escaped
      pd JEX2 >Z -> pd                             \ ","tr":"
      1 PCOL ESC[ JSON-ESCAPE  pd ESC[ >Z -> pd    \ Approved translation, escaped
      pd QQ +C [CHAR] } +C -> pd                   \ Close the pair
      INC pn
    SQLOOP
    pn IF pd [CHAR] ] +C -> pd THEN                \ Close the array
  THEN
  pd [CHAR] } +C 0 SWAP C!                         \ Close the request and terminate
;
```

Readers of the other paper will recognise `FQL`, our embedded SQL notation, first described at this conference in 2006 [5], with the implementation and experience reported in 2013 [6]. The four `+EXAMPLE-ARM` calls contribute one `UNION ALL` arm per phrase table; the results come back best-first, shortest English first among equals, and stop at `MAX-EXAMPLES`. A phrase with no significant words simply sends no examples, and the instruction alone carries the request.

Everything section 2.2 could not express, and both halves of the prefix-and-protect hack, become sentences and examples in an instruction. Readers of my other paper at this conference will recognise the shape: the capability was never the issue; the issue was having somewhere to put what we know.

3.2 The relay

The obvious implementation — every customer site calls the Anthropic API directly — puts an API key, billable to us, on hardware we do not control. So no customer site ever sees the key. Each site talks to a relay on a Micross server: `POST /translate`, `{"text","target"}` in, `{"translation"}` out, authenticated by a per-site bearer token. The relay holds the Anthropic key, the model choice, and the context. When we improve the translation instructions, every site improves on its next phrase, with no customer visit and no rebuild.

The token gets Forth-specific handling. Tracknet saves its compiled image; a token in source or left in a buffer at save time would persist into every image. So it lives in RAM only, set at start-up from a git-ignored file, and `RELAY-STARTUP` clears it unconditionally first — never trust anything a `SAVE` may have captured.

3.3 The round trip

TLS ends the raw-socket approach; `HTTPS-POST` is a thin wrapper over libcurl (30-second whole-transfer timeout, response captured into a `FILE*`). The client builds the request with a proper JSON string escaper — outbound JSON must be escaped, where before we only unescaped inbound — and the whole exchange is one word:

```
: CLAUDE-TRANSLATE { zphrase zlang zdest | phdrs pcode pmost -- f }
  FALSE                                  \ Assume failure
  TOKEN-READY? IF                        \ No site token: never try
    zphrase zlang REQ[ BUILD-REQUEST     \ Compose the request JSON
    0 Z" Content-Type: application/json"
      curl_slist_append                  \ Content type header
    Z" Authorization: Bearer " SITE-TOKEN Z+
      curl_slist_append -> phdrs         \ Bearer token header
    RELAY-URL REQ[ phdrs RESP-FILE
      HTTPS-POST -> pcode                \ POST, reply to this thread's file
    phdrs curl_slist_free_all            \ Release the header list
    pcode CURLE_OK = IF                  \ Reply received
      RESP-FILE RESP[ RESP-MAX FILE>Z DROP  \ Read the reply
      RESP[ XLAT[ PARSE-TRANSLATION IF   \ Reply carries a translation
        zphrase ZCOUNT NIP
        GROWTH-MAX * GROWTH-SLACK +
        MAXPHRASELEN MIN -> pmost        \ Longest reply worth believing
        XLAT[ ZCOUNT NIP 1 pmost BETWEEN IF \ Neither empty nor a runaway
          XLAT[ zdest ZMOVE              \ Hand the caller the translation
          DROP TRUE                      \ Everything worked
        THEN
      THEN
    THEN
  THEN
;
```

First, the shape. The word opens with `FALSE` and earns its `TRUE` at the very end, so a missing token, a transport failure, an unparseable reply, an empty reply and a runaway reply all fall through the same way: the flag stays false and the caller keeps the English. Nothing in the path can abort, because a translation is never worth stopping the application for.

Second, the plausibility bound, which exists because the characteristic failure of a language model is not corruption but excess fluency — where a broken socket gives you garbage, a chatty model gives you a paragraph *about* the translation. Section 2.3's lesson still applies in the other direction: a genuine translation may run longer than its English source, so the bound allows growth by `GROWTH-MAX` plus slack, capped at `MAXPHRASELEN`. Anything outside one to that many characters is disbelieved, and the English stays.

3.4 Testing without the server

Before the relay went live, the client was proven against a mock: sixty lines of Python speaking the identical contract on localhost, returning a marked echo. That exercised the whole Forth side — request building and escaping, bearer authentication, transport, capture, parsing, # survival, and the 401 path — with no API key and no internet. The mock found a real defect in its first hour: its JSON emitter puts a space after the colon, and our parser demanded none. When the real relay came up, the client that had only ever spoken to the mock worked against it unmodified.

The system is now in service end to end: phrases added to the application are translated through the relay as they are created.

3.5 Recording who did the translating

Automatic translation created a provenance problem. The database now holds rows translated by a human, rows translated by Google, and rows translated by Claude, and they are not equally trustworthy. So every phrase row now carries a confidence level:

```
ENUM<< CONFIDENCE
  Z" Unknown"     0 CONF_UNKNOWN   \ Untranslated (or pre-confidence data)
  Z" Google"     10 CONF_GOOGLE    \ Google gtx autotranslation
  Z" Suspect"    15 CONF_SUSPECT   \ Failed back-translation - wants a human eye
  Z" Claude"     20 CONF_CLAUDE    \ Claude relay autotranslation
  Z" Human"     100 CONF_HUMAN     \ Native speaker or manual entry
>>
```

(`ENUM<<` is the enumeration word set I proposed at this conference in 2023 [4]. It was taught to the machine through the agent directory, and the machine then used it to define the scheme by which its own translations are ranked below a human's.)

The values are ordered by trustworthiness, and the ordering is the point: a row may be overwritten by a translation of higher confidence, so the Google-era rows can be upgraded by the relay wholesale, and a human translation is never overwritten by a machine. The gaps leave room for engines not yet invented — and the newest value has already filled one. Every machine translation is now translated back to English and compared with the original: a comparison of meaning, not of wording. A failure is marked `CONF_SUSPECT` and surfaces in the Translators Aid for a human eye. The gap between 10 and 20 did not stay empty for long. Migration from approach one to approach two thereby stops being a project and becomes a query: re-translate everything below 20.

Human translations arrive in bulk through an import word, entering at `CONF_HUMAN`. When the German set was imported, Claude was asked to look it over: it noted forty typos, several spelling mistakes, and several uses of Tyrolean German. Never overwritten is not the same as beyond review. The hierarchy ranks trust, and the machine ranked below the human is nonetheless useful for proofreading above its station — the review traffic of section 4.1 flows in both directions.

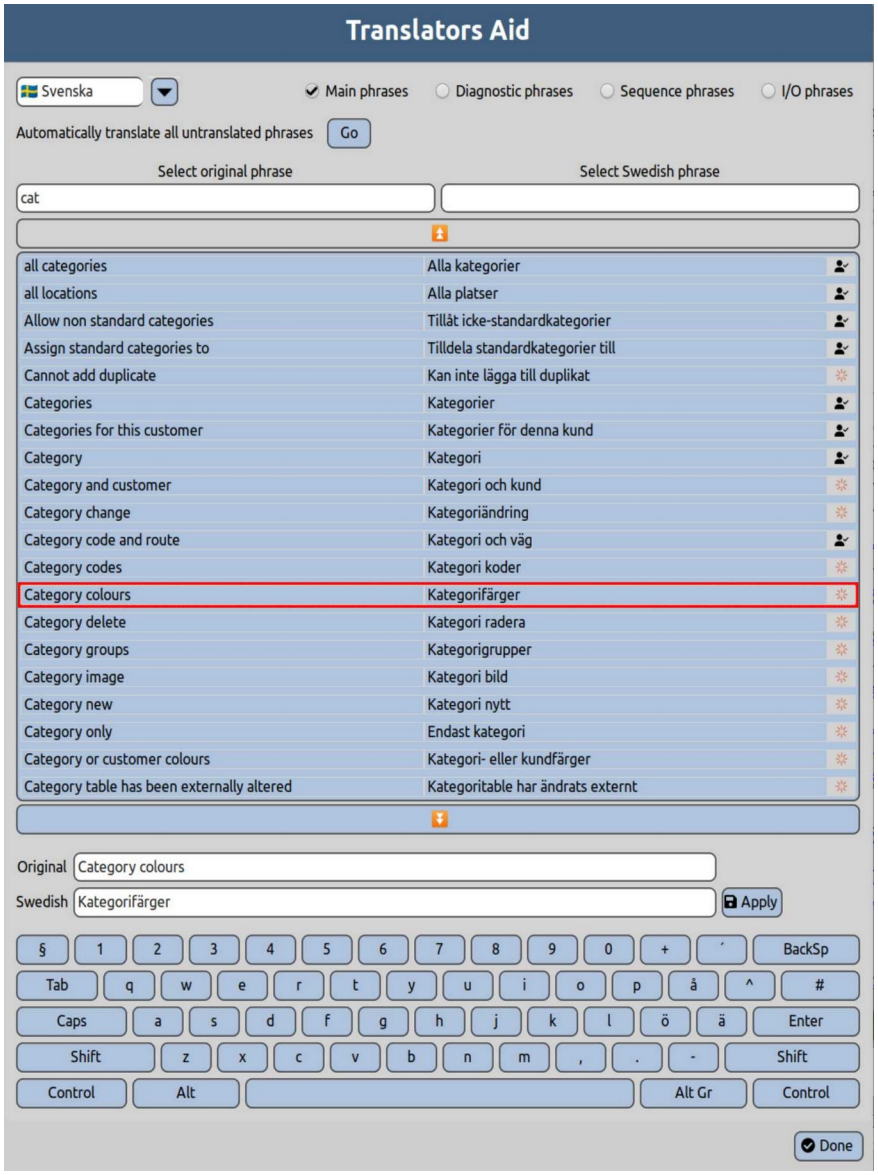


Figure 1. The Translators Aid window, in Swedish. Originally the window where every translation was typed; now the window where machine translations are corrected or enhanced. The right-hand column marks each row's provenance — human or Claude — straight from the confidence field.

The confidence level also identifies the example pool for section 3.1 — the phrases offered to the model as worked examples are those at `CONF_HUMAN`.

3.6 One phrase, three translators

The fairest comparison is a single real phrase through all three routes. From `diagphrases`, an alarm chosen because it carries a substitution code and two pieces of trade vocabulary:

> “Bag not released from storage line #”

Method	**German**	
Google	Beutel nicht aus Lagerlinie freigegeben #	Wrong noun, and the # torn from its position — the alarm loses its unit number
Claude relay	Sack nicht aus Lagerlinie # freigegeben	Glossary-correct *Sack*, # preserved in place, but generic *Lagerlinie*
Human	Sack nicht freigelassen - Lagerbahn #	The trade term *Lagerbahn*, house diagnostic style

The # row settles an account left open in section 2. `PROTECT-HASH` did its job — the character survived Google — but shielding preserves the token, not its attachment: German word order moved the sentence around it and left the unit number orphaned at the end. The model, *told* what # means, keeps it with its noun. The hack protected the characters; the context protects the meaning.

The ranking is what the confidence enum predicted: each step up gets one thing right that the step below got wrong, and the human still wins on the vocabulary that only years on the shop floor teach.

Two more:

Method	**“Overweight bag at sorting station #”**	**“Empty bag”**
Google	Übergewichtige Tasche an Sortierstation #	Leere Tasche
Claude relay	Übergewichtiger Sack an Sortierstelle #	Leerer Sack
Human	Übergewicht bei Sortierstation #	Leersack

Google’s first column is a laundry alarm about an overweight handbag. And the last column tells the whole story in five words: *Leere Tasche → Leerer Sack → Leersack.*

3.7 Does AI speak German?

My other paper at this conference asks whether AI speaks Forth. The equivalent question here was put to the translator directly: how many languages does it speak with proficiency equal to English? The answer was straight: “honestly, probably zero”, though a handful — Spanish, French, German, Portuguese, Italian — “come close enough that you’d rarely notice”, falling short only at the edges: fine register control, current slang, regional variation. A second tier (Dutch, Russian, Polish, Mandarin, Japanese, Korean, Turkish, Arabic and others) is fluent but “more likely to produce something slightly stilted or subtly off-idiom”. Beyond that lies “a long tail of lower-resource languages” where it will “sometimes generate confident-sounding nonsense”.

Self-assessment is not measurement, but it agrees with the published picture: proficiency tracks training data. Two consequences follow. First, German is a top-tier language, so section 3.6 shows the system near its best; a lower-resource column would be less flattering. Second, “confident-sounding nonsense” is precisely the failure that the plausibility bound of section 3.3 cannot catch — a fluent wrong translation is the right length. It is also exactly what the back-translation check of section 3.5 is for: what the length bound cannot see, a round trip through English often can.

Welsh is the instructive exception. By speaker count it belongs in the long tail; in practice the model handles it well, because a corpus of high-quality translations — bilingual government records among them — has been deliberately assembled for machine learning by Bangor University’s Language Technologies Unit [7]. Proficiency tracks the corpus, not the census, and a corpus is something a language community can build.

The flat value of `CONF_CLAUDE` is therefore a simplification: Claude is not one translator but a family of proficiencies under one name. The gaps in the enum exist for exactly this refinement — per-language confidence values, with the human review of section 4.1 concentrated where the tier list says trust is thinnest.

4. Comparison

	Google (gtx)	**Claude (relay)**
Transport	Raw TCP socket, HTTP GET	libcurl, HTTPS POST
Client code	~300 lines, no libraries	~230 lines + libcurl
Context	One noun phrase, smuggled into the text	Unlimited, held server-side
Substitution codes	Shield and restore around the call	An instruction: pass them through
Credentials	None	Key on relay; per-site token, RAM only
Cost	Free	About £1 per language (section 4.1)
Dependency	Undocumented endpoint that may vanish	A commercial API, and our own relay
Confidence recorded	`CONF_GOOGLE` (10)	`CONF_CLAUDE` (20)
Failure mode	English fallback	English fallback

The last row matters. Both approaches inherit the same safety property from the phrase system: the worst outcome of any failure is English text — where every phrase started.

4.1 The cost of a new language

“Free versus paid” is the obvious objection to approach two, so here is the arithmetic, measured rather than estimated. The base phrase inventory at the time of writing:

Type	**Phrases**	**Characters**
UI text (`phrases`)	1,012	14,944
Alarms (`diagphrases`)	184	6,312
Sequence steps (`seqphrases`)	747	17,394
I/O names (`IOphrases`)	773	17,816
Total	**2,716**	**56,466**

The relay currently selects Claude Haiku 4.5 — the model choice is a relay setting, and no customer site knows or cares. At current prices ($1 per million input tokens, $5 per million output, cached prompt reads about $0.10 per million), one request costs about 250 tokens of cached system prompt and glossary — byte-stable per language, so it caches — plus about 150 uncached tokens of phrase and reference examples, and about 25 tokens of translation out: **$0.0003 per phrase**.

A whole new language is therefore 2,716 requests:

	Claude relay	Human native speaker
Cost	≈ $0.85 ≈ £0.65 API	£700–£1,500 (agency £0.10–£0.15 per word of the ~10,000-word source; freelance below that; in-house native staff at a minute a phrase is about 45 hours, £900–£1,800 loaded)
Time	about 45 minutes through the relay	about a working week
Human involvement	none	all of it

A real run corroborates the estimate: 4,986 phrases across two-and-a-bit languages cost roughly $1.50 — call it £1 per language, with margin.

Roughly £1 and an hour, versus roughly £1,000 and a week: a factor of about a thousand. The comparison is not entirely fair, and section 3.6 shows why — the £1,000 buys *Lagerbahn*, and the £1 does not. But section 3.6 also shows that the machine translation is a far better starting point than the English fallback it replaces, at a price that makes "should we bother translating this language" stop being a question. The human budget is then spent where the confidence enum says it is needed: reviewing rows at `CONF_CLAUDE`, not typing 2,716 rows from nothing.

5. Conclusion

The first approach showed that the machinery of automatic translation fits comfortably in a few hundred lines of Forth and a socket — and that machinery was never the hard part. A translation engine without context produces translations without judgement, and its only channel for judgement is the text itself, which leads directly to prefix-stripping and code-shielding.

The second approach is not a different translator so much as a different interface — one with room for everything we know. The engineering around it, the queue and the retries and the fallback, carries over unchanged.

The verdict: for many common languages — and, thanks to Bangor's corpus builders, Welsh — this technique offers high-quality, finely context-sensitive translation at a tiny fraction of the cost of a professional human translator. Section 3.7 maps the tiers, the confidence enum records the difference, and the human budget moves from typing to reviewing.

There is a circularity, which regular readers will have anticipated: the Forth client that sends our phrases to Claude for translation was itself written with Claude at the keyboard, as the commit trailers admit. Whether the translator can be trusted to review its own plumbing is a question for the other paper.

References

[1] N.J. Nelson, “Internationalisation — A New Approach in Forth”, EuroForth 2019.

[2] N.J. Nelson, papers on multilingual applications, EuroForth 1995–1998, including “Internationalising WIN32 — the final steps”, EuroForth 1998.

[3] Google Cloud Translation documentation, https://cloud.google.com/translate/docs — the documented, commercial API. The free `gtx` endpoint used here is not documented.

[4] N.J. Nelson, “A proposed standard Forth style enumeration word set, using recognisers”, EuroForth 2023.

[5] N.J. Nelson, “The Nearly Invisible Database or ForthQL”, EuroForth 2006.

[6] N.J. Nelson, “Forth Query Language (FQL) — Implementation and Experience”, EuroForth 2013.

[7] Language Technologies Unit (Uned Technolegau Iaith), Canolfan Bedwyr, Bangor University — the Welsh National Language Technologies Portal, https://techiaith.cymru

EuroForth 2026
Short paper

Enhancements to a Forth style enumeration word set

N.J. Nelson B.Sc. C.Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. The word set as it stood

An enumeration is opened with `ENUM<<`, which pushes a recogniser onto the recogniser stack, and closed with `>>`, which removes it again. Between the two, any word the interpreter does not otherwise know becomes an element. A preceding value sets the enumeration counter; a preceding Zstring gives the element a description; neither is required.

```
ENUM<< TESTENUM          \ Name of the enumeration
                  AZERO  \ By default, the enumeration starts at zero
                  AONE   \ Standard Forth comments are allowed
  1 2 +           ATHREE \ Any expression with the stack effect ( ---n )
                  AFOUR  \ The enumeration increments
  Z" Customer" 11 AN11   \ Description and enumval
  Z" Category"    A12    \ Just a description
                  A13    \ Neither
>>                       \ Enumeration terminator
```

The description form is the contribution of [2], which tells a Zstring from a value by where it lives: strings are built on `SYSPAD`, nowhere near `HERE` and far above any number anyone would plausibly enumerate. That paper also reserved a cell for a phrase number, filled in at the end of the build once the database is running, and defined the first two modifiers — `ENUMPHRASE` and `ENUMDESCR` — to read them back. Preceded by one of those, an element yields that data; preceded by nothing at all, it yields its value.

It chained every element to its enumeration, and every enumeration to a list of its own, observing that how the lists might be used was not yet clear. They are what everything below walks.

2. What was missing

The gap shows up as soon as an enumeration is used for something other than naming constants. In the work described in my other paper to this conference [3], every phrase row in the database carries a confidence level recording who translated it:

```
ENUM<< CONFIDENCE
  Z" Unknown"   0 CONF_UNKNOWN \ Untranslated (or pre-confidence data)
  Z" Google"   10 CONF_GOOGLE  \ Google gtx autotranslation
  Z" Suspect"  15 CONF_SUSPECT \ Failed back-translation - wants a human eye
  Z" Claude"   20 CONF_CLAUDE  \ Claude relay autotranslation
  Z" Human"   100 CONF_HUMAN   \ Native speaker or manual entry
>>
```

The gaps are deliberate, and they get used: `CONF_SUSPECT` was slotted in at 15 after the fact, between the Google rows and the Claude ones, for a machine translation that failed back-translation verification. Nothing else had to move.

The database stores the number. The Translators Aid needs the word, to mark each row's provenance. Going from `CONF_CLAUDE` to `"Claude"` was always easy; going from the 20 that came back out of a query to `"Claude"` meant writing a `CASE` by hand, writing it again next to every other enumeration used the same way, and remembering to extend every one of them the day 15 was added. Populating a drop-down meant knowing how many elements there were and in what order they had been declared. None of that was reachable, although all of it was present in the chain.

What follows adds seven modifiers. A modifier precedes a word and changes what that word compiles or interprets to; these seven precede the enumeration name rather than an element. VFX calls the defining word `OPERATOR:`, a name that suggests something acting on operands, which is not what these do — a complaint already registered in [2]; `MODIFIER:` is a synonym for it, and is what the source now uses:

```
MODIFIER: ENUMPHRASE  \ Element: its phrase number.  n Enum: phrase number of value n
MODIFIER: ENUMDESCR   \ Element: its description z$. n Enum: description of value n
MODIFIER: ENUMCOUNT   \ Enum: the number of elements
MODIFIER: ENUMMAX     \ Enum: the largest element value
MODIFIER: ENUMMIN     \ Enum: the smallest element value
MODIFIER: ENUMMEMBER? \ n Enum: true if n is an element value
MODIFIER: ENUMEACH    \ xt Enum: execute xt ( value,zdescr--- ) for each element
```

3. Where the data lives

Before any of the queries, a structural change. Every enumeration and every element now keeps its data in separated (IDATA) space, reserved with `IRESERVE`; the word body holds nothing but the address of its block. Code and data no longer interleave, so a cache line fetched for instructions carries instructions.

```
: ENUM<< ( <name>--- )         \ Start an enumeration
  ...
  4 CELLS IRESERVE -> ENUMLIST \ Chain anchor, count, largest, smallest
  ENUMLIST 4 CELLS ERASE       \ All start zero
  ENUMLIST ,                   \ The body holds the separated block address
```

An element block is its value, its phrase number and its description, reserved to the length of that description. The chain nodes are separated too — `ENUM-LINK` reserves a link and an execution token, and makes the new element the newest:

```
: ENUM-LINK { pxt panchor | pnode -- } \ Chain a word onto an anchor, in separated space
  2 CELLS IRESERVE -> pnode            \ Separated chain node: link, xt
  panchor @ pnode !                    \ Link to the previously newest word
  pxt pnode CELL+ !                    \ Note word xt
  pnode panchor !                      \ The new word becomes the newest
;
```

The visible cost is one indirection: an element's data is now at `>BODY @` rather than `>BODY`. Everything below pays it once.

4. Statistics, folded in at definition

The other three cells of the enumeration block are the count and the largest and smallest values, and they are maintained as the enumeration is built. Each element, as it is created, folds itself in:

```
: ENUM-ADDSTATS { penum pvalue -- } \ Fold a new element value into the statistics
  penum ENUM>COUNT @ 0= IF                       \ The first element seeds
    pvalue penum ENUM>MAX !                      \ the largest
    pvalue penum ENUM>MIN !                      \ and the smallest value
  ELSE                                           \ Subsequent element
    pvalue penum ENUM>MAX @ MAX penum ENUM>MAX ! \ Track largest value
    pvalue penum ENUM>MIN @ MIN penum ENUM>MIN ! \ Track smallest value
  THEN
  1 penum ENUM>COUNT +!                          \ Count elements
;
```

Reading one is then a fetch:

```
: ENUM>COUNT ( penum---addr )   CELL+  ; \ Element count cell
: ENUM>MAX   ( penum---addr ) 2 CELLS+ ; \ Largest value cell
: ENUM>MIN   ( penum---addr ) 3 CELLS+ ; \ Smallest value cell

: ENUM-COUNT ( penum---n ) ENUM>COUNT @ ;
: ENUM-MAX   ( penum---n ) ENUM>MAX   @ ;
: ENUM-MIN   ( penum---n ) ENUM>MIN   @ ;
```

An enumeration is complete by the time anything can ask it a question, so in a definition even the fetch is unnecessary. The compiling action reads the cell there and then and lays down a literal:

```
    MOD# ENUMCOUNT   OF ENUM-COUNT CLIT,                  ENDOF \ Folded to a literal
    MOD# ENUMMAX     OF ENUM-MAX   CLIT,                  ENDOF
    MOD# ENUMMIN     OF ENUM-MIN   CLIT,                  ENDOF
    MOD# ENUMMEMBER? OF CLIT, ['] (ENUMMEMBER?) COMPILE, ENDOF \ Runtime value
```

Nothing walks the chain to count. The chain is needed only by the value lookups of section 5 and the iterator of section 6.

4.1 Sparse enumerations

It is tempting to have the largest value fall out of the count. For an enumeration that starts at zero and is left to increment, the largest value is indeed the count less one. The confidence enumeration above is not such an enumeration, and neither is any enumeration whose values mean something outside the program — bit masks, protocol codes, constants carried across from a C header:

	CONFIDENCE	**TESTENUM**
Element values	`0, 10, 15, 20, 100`	`0, 1, 3, 4, 11, 12, 13`
`ENUMCOUNT`	`5`	`7`
`ENUMMAX`	`100`	`13`
`ENUMMIN`	`0`	`0`
largest via count − 1	`4`	`6`

Table 1. Count and largest value are independent quantities.

Each is a separate cell, written as the enumeration is built and read for nothing thereafter, so there is no saving to be had by deriving one from the other, and a bug to be had by trying.

5. Membership and reverse lookup

The remaining queries take a value and search the chain for the element that carries it. A single finder does the work, and three short words dress it:

```
: (ENUMMEMBER?) { pvalue penum -- f }
  penum pvalue FINDENUMDATA 0<>
;

: (ENUMDESCR) { pvalue penum | pdata -- z$ }
  penum pvalue FINDENUMDATA -> pdata \ Find element data
  pdata IF                          \ Element exists
    pdata 2 CELLS+                  \ Its description
  ELSE                              \ No such element
    ^NULL                           \ Empty description
  THEN
;
```

The names are the point. `ENUMDESCR` before an element gives that element’s description; before the enumeration, with a value beneath it, it gives the description of the element holding that value. One name per concept, rather than one name per direction of lookup. The provenance column in the Translators Aid is `ENUMDESCR CONFIDENCE` applied to the number stored in the row, and the `CASE` that would otherwise have been written by hand — and maintained in step with the enumeration by hand — does not exist.

A membership test is not a range test. Five lies between the smallest and largest values of `TESTENUM` and is not an element of it; the question worth asking of a value arriving from outside the program is whether it names something, not whether it is between two things that do.

`FINDENUMPHRASE`, which predates this work, is now two lines forwarding to `(ENUMPHRASE)`. Its six external callers were not touched.

6. Iteration in definition order

A drop-down list, a legend, a report column: all of them want the elements in the order they were declared, and the chain links the newest first. Reversing it would disturb a structure the lookups depend on, and building a temporary array to reverse would be a poor trade for a list of five. The return stack already holds the order, if the walk is arranged to notice it:

```
: (ENUMNODE) { pnode pxt -- } \ Process from chain node pnode to the end, oldest first
  pnode IF                    \ Another element to process
    pnode @ pxt RECURSE       \ Earlier elements first (the chain links newest first)
    pnode CELL+ @ >BODY @     \ To element data
    DUP @ SWAP 2 CELLS+       \ Element value and description
    pxt EXECUTE               \ Process element ( value,zdescr--- )
  THEN
;

: (ENUMEACH) { pxt penum -- } \ Execute pxt for each element, in definition order
  penum @ pxt (ENUMNODE)
;
```

The recursion descends to the oldest element before touching anything, and executes on the way back out. Depth is bounded by the number of elements, and an enumeration long enough to trouble the return stack has troubles of its own. In use:

```
: .ELEMENT ( value,zdescr--- ) SWAP . Z$. ; \ Show one element's value and description
' .ELEMENT ENUMEACH TESTENUM               \ Show every element, in definition order
```

7. Compiling and interpreting

Everything above is reached through two `CASE` statements, one for the compiling action and one for the interpreting action, selecting on the modifier preceding the name. The modifier numbers are compiled inline by the immediate `MOD#`. Where [2] gave each modifier a constant of its own, nothing now has to be kept in step, and `.MODIFIERS` lists them by number and name. An unrecognised modifier reaches `BAD-METHOD`, which is to say it is a compile-time error rather than a run-time surprise. Adding an eighth query is one `MODIFIER:` declaration, one arm in each `CASE`, and the word itself.

8. Conclusion

The additions are small and the implementation is smaller: three cells maintained as the enumeration is built, one finder for the lookups, one recursion for the iterator, and all of it in separated space where it does not dilute the code. What they change is where knowledge about an enumeration lives. The chains built in [2] without a settled use turn out to have been the whole of it; what was implicit in them is now reachable from the name, at compile time where possible, and the hand-written `CASE` statements that used to duplicate it — and drift from it — can go.

References

[1] N.J. Nelson, “A proposed standard Forth style enumeration word set, using recognisers”, EuroForth 2023.

[2] N.J. Nelson, “Improvements to enumeration”, EuroForth 2025.

[3] N.J. Nelson, “Autotranslation within a Forth application — two different approaches”, EuroForth 2026.

EuroForth 2026
Short paper

`SNIPPET:` and `INSERT` — named source snippets in VFX Forth

N.J. Nelson B.Sc. C.Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Why a colon definition will not do

Our main Forth application, “Tracknet”, automates large commercial laundries. This includes controlling the movement of bags of laundry around the plant. Operator interfaces use GTK, the database uses MySQL. Plant behaviour is written as *sequences*: small state machines scanned every cycle. Each sequence describes itself into the database as it compiles. `SQ" <name>"` registers the sequence and `ST" <step text>"` each step description; more than that, condition words such as `?BAG`, `?SETTLED` and `?ROOM` register their own arguments, so a maintenance dialog can tell an operator in plain language exactly what a step is waiting for.

That is the whole of the difficulty. When sixteen pieces of plant need identical logic, the shared body cannot be factored into an ordinary colon definition, because a colon definition would register its descriptions once. Sixteen sequences need sixteen registrations, so the body has to be *recompiled* sixteen times. What is wanted is not a subroutine but source-level reuse: a macro.

2. Before: include a file, sixteen times

The first answer, written under time pressure for the sorting-station rewrite in May 2026, used the only off-the-peg mechanism to hand. Put the shared body in a separate file and `INCLUDE` it, mid-colon-definition, once per sequence.

One subtlety makes it work at all: a plain `INCLUDED` breaks the dictionary smudge state inside a colon definition. Opening the file and handing it to `INCLUDE-FILE` directly is safe.

```
\ Fix for INCLUDED breaking dictionary smudge state during colon definitions
: MACRO-INCLUDED ( zaddr -- ) \ Wrapper to safely include files within colon definitions
  ZCOUNT R/O OPEN-FILE THROW
  INCLUDE-FILE
;
```

At the usage site the instance parameters have to be got into the included text. They are pushed at compile time inside `[ ... ]`, and an IMMEDIATE word compiles them as literals with `POSTPONE LITERAL` before pulling in the file:

```
: SORTSTEPS ( lp station -- ) \ Evaluates lp and station at compile time
  DUP -> DESC-SS
  OVER -> DESC-LP
  SWAP POSTPONE LITERAL
  POSTPONE LITERAL
  \ Include the sorting sequence step descriptions
  Z" ../splc/Defsortsteps.fth" MACRO-INCLUDED ; IMMEDIATE

: SQ1 1 SQ" Sorting station A1" [ 1 1 ] SORTSTEPS ; : SQ1I 1 1 1 SORTYS
: SQ2 2 SQ" Sorting station A2" [ 1 2 ] SORTSTEPS ; : SQ2I 2 1 2 SORTYS
```

The shared body, in its own file, opens by consuming the two compiled literals:

```
  ( lp,ss--- )
  -> CURR-SS                              \ Current sorting station
  -> CURR-LP                              \ Current loading program
\ ******************************************************************************
  SST_FAST ST" Weight less than slow"
    COMP-LPSS SSNOTINUSE? SST_NOTINUSE ?JS \ Not in use
    COMP-LPSS SS-SLOW>                    \ Exceeds slow weight
    COMP-LPSS SSNOTINCAL AND              \ Not in calibration
    SST_SLOW ?JS
  ENDST
  ...
  ENDSQ
```

It works, and it went into production. But the shared body is exiled to a satellite file, away from the definitions that use it. Every snippet costs a file and a hard-wired path string. The parameters need the bracket-and-literal dance. And a line naming a relative path two directories away does not read like “insert the standard steps here”.

3. After: SNIPPET: and INSERT

The replacement, from August 2026:

```
SNIPPET: <name> ( in---out) <code> ;

: WORD1  ... INSERT <name> ... ;
: WORD2  ... INSERT <name> ... ;
```

`SNIPPET:` stores the source text of its body under a name. `INSERT` splices that text into a colon definition at compile time, exactly as if it had been typed there. The body lives in the same file, immediately above the definitions that use it. There is no satellite file and no path string.

3.1 Storing the body, one line at a time

The line structure has to be preserved. A `\` comment runs to the end of a *line*, so flattening the body into a single string would let the first `\` swallow everything after it. This is precisely what the old method got for nothing: `INCLUDE-FILE` handles lines. A snippet has to keep them itself, so each line is stored separately — a length cell, the characters, aligned — with a `-1` sentinel at the end.

```
: SNIPPET-LINE, { paddr plen -- } \ Store one snippet source line in the dictionary
  plen ,                          \ Line length
  HERE plen ALLOT                 \ Reserve the line space
  paddr SWAP plen MOVE            \ Store the text
  ALIGN                           \ Align for the next length cell
;
```

3.2 Finding the terminator

The body ends at a bare `;` token. Finding it means tokenising each line and ignoring any `;` that sits inside a comment: a `\` token ends the scan for that line, and a `(` token skips to the closing `)`. Only a single-character token counts, so a word such as `(SNIPPET-LINE)` or `;M` in the body cannot false-trigger.

```
\ Offset past a bare ; token in a line, 0 if none
: SNIPPET-; { paddr plen | ptok pcnt pres -- n }
  0 -> pres                       \ Assume no terminator
  paddr plen                      \ Scan the whole line
  BEGIN
    -LEADING -> pcnt -> ptok      \ Start of the next token, tabs treated as spaces
    pcnt 0<>  pres 0=  AND        \ A token left and the terminator not yet found
  WHILE
    ptok pcnt BL SCAN             \ Rest of the line beyond the token
    pcnt OVER - 1 = IF            \ Single character token
      ptok C@ [CHAR] ; = IF       \ The terminator itself
        ptok paddr - 1+ -> pres   \ Offset past the terminator
      THEN
      ptok C@ [CHAR] \ = IF       \ Comment runs to the end of the line
        2DROP ptok 0              \ Nothing more to scan on this line
      THEN
      ptok C@ [CHAR] ( = IF       \ Bracket comment
        [CHAR] ) SCAN DUP IF      \ Closing bracket present
          1 /STRING               \ Step over the closing bracket
        THEN
      THEN
    THEN
  REPEAT                          \ Next token
  pres                            \ Offset past the terminator, 0 if none
;
```

The scanner is built on VFX's `-LEADING`, `SKIP` and `SCAN`, which treat a tab as a space when scanning for `BL`. That matters more than it sounds: a first hand-rolled version compared characters against `BL` alone, and house source is tab-indented, so a tab-delimited `;` would have been missed. The kernel words made the scanner both shorter and correct.

```
\ Capture one line of snippet source, true at the terminating ;
: (SNIPPET-LINE) ( ---f )
  SOURCE >IN @ /STRING { paddr plen | pterm -- } \ Remainder of the current source line
  paddr plen SNIPPET-; -> pterm   \ Terminator position, if present
  pterm IF                        \ Terminator on this line
    paddr pterm 1- SNIPPET-LINE, \ Store the text before the terminator
    pterm >IN +!                  \ Consume the source through the terminator
  ELSE                            \ Whole line belongs to the macro
    paddr plen SNIPPET-LINE,      \ Store the whole line
  THEN
  pterm 0<>                       \ True when the terminator was seen
;
```

3.3 Defining and splicing

`SNIPPET:` is then a loop over lines until the terminator is seen, with `REFILL` moving to the next source line:

```
: SNIPPET: ( "name" --- ) \ Define a named source snippet, body up to a bare ; token
  CREATE                          \ Name the snippet
  BEGIN
    (SNIPPET-LINE) 0=             \ Capture a line, done at the terminator
  WHILE
    REFILL 0= ABORT" SNIPPET: missing ;" \ Move to the next source line
  REPEAT                          \ Next snippet line
  -1 ,                            \ End of snippet sentinel
;
```

And `INSERT` is IMMEDIATE: it ticks the name and `EVALUATE`s the stored lines one at a time. `EVALUATE` runs in the current `STATE`, so inside a colon definition the snippet text compiles — identical semantics to the include, without the file.

```
: INSERT ( "name" --- )           \ Splice a snippet's source into the current definition
  ' >BODY { pline | plen -- }     \ Snippet line storage
  BEGIN
    pline @ -> plen               \ Length of the next line
    plen -1 <>                    \ Until the end sentinel
  WHILE
    pline CELL+ plen EVALUATE     \ Compile the line in the current context
    pline CELL+ plen + ALIGNED -> pline \ Next line
  REPEAT                          \ Next snippet line
; IMMEDIATE
```

Everything happens at compile time. The compiled code is byte-for-byte what writing the body out longhand would produce, and the VFX optimiser sees ordinary inline source. The only cost is dictionary space for the stored text, a few hundred bytes per snippet.

4. Compile-time parameters: the snippet reads its own literal

This part matured after the facility was first written, and it is the part most likely to interest a Forth audience.

Each use of the snippet needs its instance number. The shells pass it in the simplest way possible — a plain literal ahead of `INSERT`, consumed at run time by the snippet's opening `-> CURR-PR`:

```
: SQ50 50 SQ" ..." 1 INSERT PRSTEPS ;
```

But the describing words need that number at *compile* time. The conditions in the step body must compile literal bag and input numbers so that the self-description machinery can record them. Tracknet's describing words discover their argument with `SQD-GETLIT`, a deliberate kludge that inspects the machine code just compiled: if the preceding compilation was a literal — directly, or through the optimiser's constant folding — it reads the value out of the code stream without disturbing it, and the word registers a description such as "Bag 59 present". An argument calculated at run time reads as −1 and registers only "(Calculated)".

The snippet turns this to its advantage. Its first line reads the shell's own literal back out of the compiled code and saves it in a compile-time global:

```
  [ SQD-GETLIT -> DESC-PR ] \ Pressure relief number for the description literals
  -> CURR-PR                \ Current pressure relief
```

`DESC-PR` then drives a family of IMMEDIATE literal generators, one per instance-varying parameter — source bag, destination bag, storage line, late-arrival diagnostic and the three inputs — each a table lookup folded to a literal at compile time. The repeated trailing comments are elided here; each reads "Compile the … as a literal":

```
: CURR-PRSRC      ( --- ) DESC-PR PR-SRC   POSTPONE LITERAL ; IMMEDIATE
: CURR-PRDEST     ( --- ) DESC-PR PR-DEST  POSTPONE LITERAL ; IMMEDIATE
: CURR-PRLINE     ( --- ) DESC-PR PR-LINE  POSTPONE LITERAL ; IMMEDIATE
: CURR-PRDIAG     ( --- ) DESC-PR PR-DIAG  POSTPONE LITERAL ; IMMEDIATE
: CURR-PRSRCX     ( --- ) DESC-PR PR-SRCX  POSTPONE LITERAL ; IMMEDIATE
: CURR-PRROOMX    ( --- ) DESC-PR PR-ROOMX POSTPONE LITERAL ; IMMEDIATE
: CURR-PRDESTX    ( --- ) DESC-PR PR-DESTX POSTPONE LITERAL ; IMMEDIATE
```

So each of the sixteen compilations of the snippet emits *its own* bag and input numbers as literals, every describing word records real values, and the maintenance dialog shows "Bag at start of storage line 5 stage 2" rather than "(Calculated)" — all while the source is written once. The lookup tables are built with `LOOKUP<<`, a sibling facility from the same week: entries accumulate on the data stack by ordinary interpretation, and the general terminator `>>` counts and stores them.

5. A real example: sixteen pressure reliefs

The first production use was at the St. Joris site in Belgium. Storage lines 1–4 are divided into two stages, lines 5–10 into three; between every pair of adjacent stages sits a *pressure relief*, a stop that releases one bag from the end of one stage to the start of the next. All sixteen need the same two-step sequence. Only the bag numbers, storage line, diagnostic phrase and I/O differ, and those live in tables indexed by pressure relief number.

Under the old method that meant a satellite file and the literal dance:

```
: PRSTEPS ( pr--- ) \ Compile the standard pressure relief steps for one sequence
  POSTPONE LITERAL  \ Compile the pressure relief number
  Z" ../splc/Defprsteps.fth" MACRO-INCLUDED \ Include the standard step definitions
; IMMEDIATE

: SQ50 50 SQ" Pressure relief" [  1 ] PRSTEPS ; : SQ50I 50  1 PRYS ;
: SQ51 51 SQ" Pressure relief" [  2 ] PRSTEPS ; : SQ51I 51  2 PRYS ;
```

The snippet now sits directly above its sixteen users:

```
SNIPPET: PRSTEPS ( pr--- )   \ The standard pressure relief steps
  [ SQD-GETLIT -> DESC-PR ]  \ Pressure relief number for the description literals
  -> CURR-PR                 \ Current pressure relief
\ ******************************************************************************
  0 ST" Wait for bag ready"
    CURR-PRSRC ?BAG          \ Bag at the source stage start
    CURR-PRDEST ?NOBAG AND   \ Space at the destination stage end
    CURR-PRSRCX ?SETTLED AND \ Bag steady against the pressure relief stop
    CURR-PRROOMX ?ROOM AND   \ Room on the destination stage
    ?NS                      \ All conditions met, release the bag
  ENDST
\ ******************************************************************************
  1 ST" Bag runs to storage line"
    \ Bag arrived within time, else the late diagnostic
    CURR-PRDESTX X CURR-PRDIAG CURR-PRLINE 10SD IF
      CURR-PRSRC CURR-PRDEST MOVBAG \ Move the bag image between the stages
      RS                     \ Movement complete, back to the start
    THEN
  ENDST
\ ******************************************************************************
  ENDSQ
;

: PRYS { pseq ppr -- }       \ Outputs for one pressure relief sequence
  ppr PR-STOPY ?DUP IF       \ Stop output allocated
    pseq SS 1 = SWAP Y       \ Stop open while the bag is released
  THEN
;

: SQ50 50 SQ" Pressure relief storage line 1 stage 2"   1 INSERT PRSTEPS ;
  : SQ50I 50  1 PRYS ; DESCRIBE-YS
: SQ51 51 SQ" Pressure relief storage line 2 stage 2"   2 INSERT PRSTEPS ;
  : SQ51I 51  2 PRYS ; DESCRIBE-YS
: SQ52 52 SQ" Pressure relief storage line 3 stage 2"   3 INSERT PRSTEPS ;
  : SQ52I 52  3 PRYS ; DESCRIBE-YS
\ ... through ...
: SQ65 65 SQ" Pressure relief storage line 10 stage 3" 16 INSERT PRSTEPS ;
  : SQ65I 65 16 PRYS ; DESCRIBE-YS
```

The whole sixteen-sequence feature is seven small lookup tables, seven one-line literal generators, one snippet, one shared output word and sixteen one-line shells. Commissioning the real I/O is purely a lookup-table edit.

Two details are worth drawing out. Sequence *names* are per-instance — "Pressure relief storage line 5 stage 3" — so the operator's selection dialog is usable, while step *texts* are shared verbatim across all sixteen, so the translated phrase tables gain only the unavoidable rows. And the feature runs two describing regimes at once: conditions describe at compile time, through the literals of section 4, whereas outputs describe at execution time. `DESCRIBE-YS` after each I-word temporarily points the `Y` DEFER at a describing action and runs the word once per step, so it works through the shared `PRYS` with run-time values and needs no literals at all.

6. The naming story

The facility was designed as `MACRO:` and `INSERT`. On first compile, Tracknet's redefinition guard stopped it dead:

```
MACRO:  redefinition not allowed
```

VFX Forth, it turns out, already owns a word `macro:`. It is undocumented — it does not appear in the manual [1] — and lives in the kernel's `PreAsm.fth`, where it defines *assembler* macros: reusable chunks of assembler source for use inside `CODE ... END-CODE`, placed in the `asm-access` vocabulary, with a prologue and epilogue that save and restore the assembler's operand-prefix state. Internal machinery for MPE's own code generator — and, pleasingly, the same idea: source-text reuse, welded to the assembler.

Tracknet hooks `RedefHook` to make any redefinition a hard error, chiefly as protection against duplicate glade widget ids becoming duplicate Forth names:

```
\ Do not allow inadvertant redefinitions (e.g. duplicate id names in glade and/or
\ settings)
: NOREDEFS ( --- )
  CR LASTNAMEFOUND .NAME ."  redefinition not allowed" ABORT
; ASSIGN NOREDEFS TO-DO REDEFHOOK
```

The guard was doing its job. Rather than bypass it and silently bury a kernel word, the facility was renamed `SNIPPET:`.

7. Discussion

Portability. Almost all of it is standard [2]: `SOURCE`, `>IN`, `/STRING`, `REFILL`, `EVALUATE`, `CREATE`, `ALIGN` and `ALIGNED`. The VFX-specific parts are `-LEADING`, `SKIP` and `SCAN`, which are trivially replaceable, the named locals `{ ... }`, and the house `->` (a synonym of `TO`). The technique should port to any system whose `EVALUATE` respects `STATE` and whose `REFILL` works during a defining word. `SQD-GETLIT` is emphatically not portable — it decodes x86-64 code bytes — but the snippet works without it. It is the bridge to Tracknet's self-description machinery, not part of the macro facility.

Limitations. The terminator must be a bare, whitespace-delimited `;` outside comments. A `;` inside a string literal in the body would false-trigger; in practice this does not arise, since strings containing a lone semicolon are rare. Nesting — `INSERT` inside a snippet body — is expected to work when the inserted name is on the same stored line, but is untested.

Why not one big string? Comments, again. `\` runs to the end of a line, and `EVALUATE` treats its whole string as one line. Line-by-line storage is the essential trick, and it is exactly what `INCLUDE-FILE` supplied in the old method — now without the file.

8. Conclusion

Seventy lines, most of them comment, replace a mechanism that needed a satellite file, a hard-wired path and a compile-time literal dance. The gain is not performance — the generated code was always identical — but locality: the shared body now sits where a reader expects it, beside the sixteen definitions that use it, and `INSERT PRSTEPS` says what it does. That the snippet can also read back the literal its caller compiled, and hand it to the description machinery, was not in the original design; it fell out of having the body in ordinary source rather than in a file. The old method remains in the tree in `Sortsequences.fth`, pending conversion.

References

[1] MPE Ltd, VFX Forth for Linux, User Manual.

[2] Forth 200x Standards Committee, Forth 2012 Standard, forth-standard.org.

The problem of false positives in statistical tests: countering mistaken intuitions with Monte Carlo simulations.

Bill Stoddart
w.j.stoddart@gmail.com

August 27, 2026

Abstract

The statistical problem of "false positive" conclusions being drawn from information arising due to the probability of false positives being directly proportional to the rarity of a phenomenon. Programming a Monte Carlo simulation forces all relevant assumptions to be clearly defined and can provides us with evidence from multiple simulations. We show how such methods can be implemented in RVM_FORTH, our reversible Forth.

1 Introduction

In 2001 the Royal Statistical Society wrote to the UK government (in the form of the Lord Chancellor) to express concern over the misrepresentation of statistical information in criminal trials. The concern was prompted by the murder conviction of Sally Clark for murder after her two babies died of cot death. They had multiple concerns, but the one that concerns us here is the problem of "false positives" that arises in the investigation of statistically rare phenomena.

2 Case Study

We illustrate the frequency of false positive conclusions by means of a simulated medical trial. We assume a disease currently infects 1% of a certain population. We have a test for detecting the disease with an accuracy[1] of 99%. If a person chosen at random from the population tests positive, what is the probability that they have the disease?

For our Monte Carlo trial, we will assume we have a random sample of size 10,000 from our population. The individuals in our sample are numbered 0

[1]To simplify our presentation we do not distinguish between sensibility and specificity of test performance.

to 9999. We assume each individual has a probability of 1% of having the disease, and we generate the set of infected individuals by running through all members of our SAMPLE and placing each individual in the infected set with a probability of 1%.

When we test each individual for the disease, there are 4 possible results, which we identify with the following codes: 1 indicates true +ve, 2 false -ve, 3 true -ve, 4 false +ve.

Any individual from the infected set will give a true positive result with probability 99%, and a false negative result with probability 1%.

Any other individual will have a true negative result with a probability of 99% and a false positive result with a probability of 1%.

In out implementation, as we assign these codes we build a sequence of length 10000 that records the code associated with each individual. To retrieve the number of individuals associated with each code we invert the sequence and take the relational image of 1, 2, 3 and 4 respectively.

To illustrate how this would work for a sequence of length 7, say $S = [\, 3 \,,\, 1 \,,\, 3 \,,\, 3 \,,\, 4 \,,\, 3 \,,\, 1 \,]$

As a set, S can be described as follows:

$S = \{\, 1 \mapsto 3 \,,\, 2 \mapsto 1 \,,\, 3 \mapsto 3 \,,\, 4 \mapsto 3 \,,\, 5 \mapsto 4 \,,\, 6 \mapsto 3 \,,\, 7 \mapsto 1 \,\}$

When inverted this becomes:

$\sim S = \{\, 3 \mapsto 1 \,,\, 1 \mapsto 2 \,,\, 3 \mapsto 3 \,,\, 3 \mapsto 4 \,,\, 4 \mapsto 5 \,,\, 3 \mapsto 6 \,,\, 1 \mapsto 7 \,\}$

Now the number of true negatives, say, (code 3) is given by counting how many values 3 maps to, which we do by checking the size of the set $\{\, 1 \,,\, 3 \,,\, 4 \,,\, 6 \,\}$

3 Some code

3.1 Generating the set of infected individuals

The code for this is:

```
: SI ( -- s,
  generate the set of infected individuals from the sample )
  INT { <RUN SAMPLE CHOICE IRATE ?% --> RUN> } ;
```

`INT { .. }` opens an integer set declaration. `INT` is defined as an empty integer set, from which `{` can extract the information to create a new integer set. `}` performs the housekeeping required to terminate the construction of a set.

Code between `<RUN ...RUN>` encloses a generate and test section which produces the elements of the new set.

`SAMPLE CHOICE` deterministically chooses values from the sample set 0..9999. Each time code reverses to this point a new value is taken. When no more

values are available the generate and test section terminates.

IRATE is the population infection rate expressed as a percentage, say 1 per cent. ?% is a probabilistic flag which depends on IRATE and has a 1 per cent chance of being true. --> is a naked guard which consumes the flag and allows forward continuation if the flag is true. In this case the value previous generated by CHOICE is added to the set being constructed. Otherwise code reverses back to CHOICE which generates another value if one is available.

4 Allocating a test status to each individual

```
: TEST ( n -- 1|2|3|4,
   1 indicates true +ve, 2 false -ve, 3 true -ve, 4 false +ve )
   INFECTED IN
   IF TA ?%
     IF 1 ELSE 2 THEN
     ELSE TA ?% IF 3 ELSE 4 THEN
   THEN ;
```

Before this code runs, the set of infected individuals, generated by SI has been moved to the value INFECTED.

TA is a value which holds the test accuracy as a percentage. ?% is a probabilistic flag and TA ?% has a probability of TA per cent of being true.

5 Running a simulation

```
: T  ( run simulation )
  SI to INFECTED
  INT [ <RUN SAMPLE CHOICE TEST RUN> ] ~
  DUP 1 ELEMENT-IMAGE CARD to TRUE+
  DUP 2 ELEMENT-IMAGE CARD to FALSE-
  DUP 3 ELEMENT-IMAGE CARD to TRUE-
      4 ELEMENT-IMAGE CARD to FALSE+
      REPORT ;
```

Here the generate and test section generates a sequence rather than a set, as indicated by the use of sequence brackets [..].

The sequence is inverted with ~.

We then generate element images corresponding to our test status values 1, 2, 3 and 4. The size of these sets (obtained with CARD) tells us how many individuals of each category we have.

6 Some results

Here are a few simulations. The command T runs a single simulation. Do these figures change your intuitive estimate of the probability of an individual having the disease if they have tested positive?

```
T+ = 108  F- = 1   T- = 9794  F+ = 97 ok
T
T+ = 115  F- = 0   T- = 9794  F+ = 91 ok
T
T+ = 95  F- = 2   T- = 9809  F+ = 94 ok
T
T+ = 105  F- = 1   T- = 9802  F+ = 92 ok
T
T+ = 95  F- = 0   T- = 9817  F+ = 88 ok
T
T+ = 83  F- = 1   T- = 9821  F+ = 95 ok
T
T+ = 92  F- = 1   T- = 9810  F+ = 97 ok
T
T+ = 97  F- = 1   T- = 9822  F+ = 80 ok
T
T+ = 101  F- = 0   T- = 9802  F+ = 97 ok
T
T+ = 86  F- = 1   T- = 9806  F+ = 107 ok
T
T+ = 96  F- = 0   T- = 9795  F+ = 109 ok
T
T+ = 90  F- = 0   T- = 9794  F+ = 116 ok
T
T+ = 101  F- = 2   T- = 9778  F+ = 119 ok
T
T+ = 109  F- = 3   T- = 9811  F+ = 77 ok
```

7 Conclusions

A few lines of `RVM_FORTH`, our reversible Forth implementation, are enough to create a Monte Carlo simulation of a clinical trial.

One way in which such simulations can be useful is in showing that in any statistical test the probability of a false positive conclusion is directly proportional to rarity of the phenomena being tested, independently of the reliability of the test used.

This seems to have been a difficult fact for expert witnesses in criminal trials to fully comprehend. We have cited the case of the mother convicted of murdering her children when they died of natural cot death. Cot death is rare, but mothers murdering children, in the absence of possible explanations, is very rare.

Other examples from UK legal history include the current situation of Lucy Letby, convicted for the murder of babies in her nursing care, and the actions taken (without trial) to remove children from parents during the Cleveland "child abuse scandal".

Acknowledgement. The relationship between false positives, condition incidence, and test sensitivity was brought to my attention by my grandson Tobias Ageass Stoddart who had noticed it during his Baccalaureate studies.

The author is grateful to an artificial colleague for their careful reading and correction of the paper.

Russell's Paradox Disarmed
42nd Euroforth, Cardiff, September 2026

Bill Stoddart
w.j.stoddart@gmail.com

August 25, 2026

Abstract

In 1902 Bertrand Russell wrote to Gottlob Frege to point out a paradox which destroyed the consistency of the latter's formalisation of set theory, of which the second volume was about to be published. This exchange caused the abandonment of the most straightforward formulation of set comprehension, in which every logical predicate defines the set of objects it satisfies. This not only complicated the theory of sets, but moved it further from logic. In our reversible Forth we use a formal semantics based on Eric Hehner's Bunch Theory, a variant of Set Theory in which Russell's Paradox, and paradoxes in general, have a different interpretation which doesn't undermine the theory's foundations, and allows us to put forward some speculative ideas.

Contents

1 Introduction

2 Russell's Paradox

3 The Barber's Paradox

4 Descriptions

5 Recovering the simple comprehension law of Cantor's Paradise

6 Other Paradoxes in Naive Set Theory

7 Combining Typed and Untyped Reasoning in a Single Theory

8 Conclusions and Future work

1 Introduction

In the early days of set theory, that is before Russell formulated his paradox, sets were defined by a simple "comprehension law" which linked any logical predicate to the set of elements that satisfied it.

For example if our predicate is $x = 1 \ \vee \ x = 2$ ($x = 1$ *or* $x = 2$) the corresponding set is $\{ 1, 2 \}$ which is written with this simple set comprehension notation as:

$\{ x \mid x = 1 \vee x = 2 \}$

And this generalises to the set comprehension law in which E is an expression and $P(x)$ is a predicate on x :

$E \in \{ x \mid P(x) \} \Leftrightarrow P(E)$

The universal set[1] which contains all elements is defined by:

$U \mathrel{\widehat{=}} \{ x \mid \mathit{true} \}$

This version of set theory, which was proposed by Cantor and Frege, appeared to give a highly satisfactory state of affairs in which set theory was simple to formulate and sets (which formed the foundations of mathematics) were directly related to logical predicates, which accorded with the intuition that logic was the basis of mathematical theory. This period was later referred to as "Cantor's Paradise".

2 Russell's Paradox

To formulate his paradox Russell first noted that a set might or might not belong to itself. As an example of a set that does not belong to itself consider $\mathbb{N}$, the set of "numbers" (in everyday terms, the set of non-negative whole numbers).

$\mathbb{N} = \{ 0, 1, 2, ... \}$

The set $\mathbb{N}$ is not itself a number, so doesn't belong to itself: $\mathbb{N} \notin \mathbb{N}$.

The empty set is another example of a set that does not belong to itself.

However, if we assume we can define the universal set U, that would be an example of a set that belongs to itself.

Before we introduce Russell's paradox, consider for a moment what kind of answer we might be satisfied with if we ask whether an element E belongs to a set defined in terms of a predicate P:

$E \in \{ x \mid P(x) \}$?

The answer will be "yes" if $P(E)$ is true and "no" if $P(E)$ is false.

Russell proposed the set of all sets that don't belong to themselves:

[1] We discuss later whether such a set exists

$R \mathrel{\widehat{=}} \{ x \mid \neg (x \in x) \}$

Then asks us to consider whether R belongs to itself. By our law of comprehension we obtain:

$R \in R \Leftrightarrow \neg (R \in R)$.

This is not a yes or no answer, instead it says "it does if and only if it doesn't".

Let us examine carefully how this causes our set theory to collapse. We will need two logical inference rules: MODUS PONENS says that from a and $a \Rightarrow b$ we can conclude b. From REDUCTIO AD ABSURDUM, we can prove a by assuming $\neg a$ and then proving a contradiction, i.e. proving both b and $\neg b$ for some b of our choice.

First we expand $R \in R \Leftrightarrow \neg (R \in R)$ into its two component parts: (1) $R \in R \Rightarrow \neg (R \in R)$ and (2) $\neg (R \in R) \Rightarrow R \in R$.

Now we can prove $R \in R$ by REDUCTIO AD ABSURDUM. We assume the opposite of our goal, that is $\neg (R \in R)$. Now we have to prove two contradictory statements. These will be $R \in R$ and $\neg (R \in R)$. We prove the first of these by MODUS PONENS from $\neg (R \in R)$ and $\neg (R \in R) \Rightarrow R \in R$. That establishes $R \in R$. We don't need further proof to establish $\neg (R \in R)$, as we are already reasoning under this assumption.

By a similar argument we can prove $\neg (R \in R)$. Now we have a suitable b and $\neg b$ which we can use to prove anything we like. A system in which anything can be proved is unsound, our theory is unusable.

3 The Barber's Paradox

To describe the idea of paradox to a wider audience, Russell proposed a village in which all men are clean shaven, and in which the barber, who is a man, shaves those men (and precisely those men) who do not shave themselves. Who shaves the barber?

It can't be that the barber shaves himself, because we know he shaves just those men who do not shave themselves.

But if he does not shave himself we know he must be shaved by the barber.

Although our description appears to say something about a village, that village cannot exist, its description is over-constrained.

4 Descriptions

We might wonder why Russell's description of the set R caused mathematics to be reformulated, whilst his description of the village and barber leaves natural language unscathed.

We propose that the reason is that in mathematics as it is conventionally formulated, e.g. for Abrial's B-method where we have a typed set theory underpinned by classical predicate logic with equality, any expression represents an element, a singular mathematical object such as a number or a set, even where it is clear that no such object exists. For example, in this formalism we can prove there exists some number equal to $1/0$. That's an example of an "undefined term", and the normal approach is to deal with it by saying ok, it exists, but we can never know what it is. This is in contrast to natural language descriptions which can represent pluralities or nothing as well as individuals.

So the problem, in a nutshell, is that a paradox is an over-constrained description which describes nothing, but, in conventional mathematics, expressions always correspond to an element.

We can escape from Russell's Paradox if we allow mathematical terms to describe nothing or pluralities (non-deterministic terms) as well as elements.

Eric Hehner's theory of bunches[Heh93] allows this by giving a mathematical meaning to the contents of sets. If we consider the set $\{1, 2\}$ we can extract its contents with the unpack operator $\sim$, thus $\sim\{1, 2\} = 1, 2$. One way we can interpret $1, 2$ is as a non-deterministic term that might be 1 or might be 2.

But what meaning can we give to the contents of the empty set? This literally is nothing. In Bunch Theory, nothing is a fully paid up member of our conceptual universe, we refer to it as *null* and define it as:

$null \mathrel{\widehat{=}} \sim\{\ \}$.

In Bunch theory we abandon the fiction that $1/0$ is some unknowable number, we just admit it doesn't exist and say $1/0 = null$. Since in classical logic we can prove $\exists\, x \bullet x = 1/0$ this requires some tweaks in the inference rules for quantifiers in our logic, and these are detailed in [DMSZ24].

We can similarly dispose of the current king of France, who gave Russell quite a lot of trouble when he pondered the truth of "the current king of France is bald".

5 Recovering the simple comprehension law of Cantor's Paradise

We can recover a version of Cantor's simple set comprehension axiom for $E \in S$ by limiting ourselves to the case where both E and S are elements. We use $\delta(E)$ to say that E represents an element, and propose the following law.

$\delta(E) \wedge \delta(\{\ x \mid P(x)\ \}) \Rightarrow (\ E \in \{\ x \mid P(x)\ \} \Leftrightarrow P(E)\ )$

Let's try that out with Russell's R. We get:

$(\ \delta(R) \wedge \delta(\{\ x \mid \neg\,(x \in x)\ \}\ ) \Rightarrow (\ R \in \{\ x \mid \neg\,(x \in x)\ \}\ ) \Leftrightarrow \neg\ (R \in R)\ )$

and since by definition $R = \{\ x \mid \neg\,(x \in x)\ \}$ we can write this as:

$\delta(R) \ \wedge \ \delta(R) \ \Rightarrow \ (\ R \in R \ \Leftrightarrow \ \neg \ (R \in R) \ )$ which is equivalent to:

$\delta(R) \ \Rightarrow \ \mathit{false}$ which by contrapositive argument gives:

$\neg \ \delta(R)$.

Thus attempting to apply our set comprehension rule to R tells us that R is not an element and therefore the rule cannot be applied.

6 Other Paradoxes in Naive Set Theory

For a survey of the paradoxes of naive set theory see the report of Justin Miller [Mil01].

Russell's paradox is not the only antinomy in naive set theory, but since we have an escape hatch, in the form of the *null* bunch, their existence does not threaten the soundness of our theory, but we might still ask whether they cause enough inconvenience, in the sense of allowing us to write down descriptions of sets we *think* exist when they do not, or whether, for most practical purposes, they will not trouble us. Here, a key point is that such paradoxes often involve infinite sets, and, for practical use, we may have to use some predicates directly without supposing they can be used to describe a set.

One paradox, due to Cantor, is the paradox of the universal set. If we take any set S we can always construct a larger set $\mathbb{P}(S)$[2]. But can there be a larger set than $U \ \widehat{=} \ \{x \ | \ \mathit{true} \ \}$? Obviously not, so the universal set does not exist.

7 Combining Typed and Untyped Reasoning in a Single Theory

In the 40th EuroForth we presented a semantic theory of values for a stack based language using a typed version of Bunch Theory. However, a typed approach does impose limitations since Forth programs may interpret the same primitive data as representing more than one abstract value. In this section we discuss how we can combine our existing ability to reason about homogeneous (i.e. typed) sets with our newfound freedom to formulate an untyped theory. Indeed, the way we do this provides an example of both the power of untyped set theory and the risks associated with its use.

First let us try to provide new axioms that define an untyped set T that contains all the homogeneous sets from our typed set theory.

Name	Rule	Condition
Given Set	$S \ \in \ T$	S a given set
Power Set	$\mathbb{P}(S) \ \in \ T$	$S \ \in \ T$
Product set	$S_1 \ \times \ S_2 \ \in \ T$	$S_1, S_2 \ \in \ T$

[2]See the appendix for discussion and proof

Have we really defined T ? Could we not take T and generate a bigger set by re-applying our type constructors?

Let's try a different approach which defines a chain of sets. Let T_0 be the set of given sets. Now let T_1 contain the members of T_0 plus any sets we can construct from them using powerset and set products.

Now to continue in this way we define two generators f and g and use them to define a chain of sets containing an increasing number of types, and say some t is a type if it is a member of a set in this chain:

$$f(T_n) = \{ x \mid \exists y \bullet y \in T_n \land x = \mathbb{P}(y) \}$$
$$g(T_n) = \{ x \mid \exists y_1, y_2 \in T_n \land x = y_1 \times y_2 \}$$
$$T_{n+1} = T_n \cup f(T_n) \cup g(T_n)$$

Now we can give a typed set comprehension rule which defines membership for sets that are carved out of one of the homogeneous sets in our set T of types.

Name	Rule	Condition
Typed Compr'n	$E \in \{ x \mid x \in S \land P(x) \} \Leftrightarrow E \in S \land P(E)$	$\exists n \bullet S \in T_n$

8 Conclusions and Future work

Conventional mathematics is a theory of "elements", and insists that every expression must represent an element, i.e a singular entity such as a number or a set. Bunch theory generalises this to give a theory of unpackaged collections. This includes the empty collection *null*, which literally represents nothing.

Russell gave us the Barber's Paradox to illustrate how a paradox arises from an over-constrained description, and this is true of paradoxes in general. Our ability to represent an over-constrained description as *null* provides the escape hatch that allows us to disarm them.

We can recover the classical axiom for $E \in S$ so long as we insist that E and S are elements. In the case of Russell's putative set R, we can use the same axiom to show R is *not* an element.

Freeing up set theory from constraints (such as type constraints) which prevent paradoxes allows the maximum freedom of description, as reflected in the term "Cantor's Paradise". However, it also allows us to "define" sets which do not exist. Their non existence does not threaten our theory, but can render a particular application of the theory incorrect.

Our current bunch theory is strictly typed. Types in our current theory are maximal homogeneous sets which are either provided as "given sets" or produced using set constructors. The unconstrained set comprehension discussed in this paper gives us an untyped set theory. To move to an untyped theory whilst retaining the ability to reason about typed sets we attempt to construct an untyped set of all required types, and use this to constrain an additional *typed* set comprehension rule. However, we must take care not to overreach ourselves

by claiming we can directly define the set of all our possible types, as this also leads to paradox. Instead, we define an infinite ascending chain of sets which provide increasingly rich sets of types and say t is a type if it is included in a member of this chain.

Acknowledgements

The author is grateful for discussions with Rick Hehner and Steve Dunne and with an artificial colleague and also for the careful reading of the text by an artificial colleague.

References

[DMSZ24] S E Dunne, C Mu, W J Stoddart, and F Zeyda. Bunch theory: axioms, logic, applications and model. *Journal of Logical and Algebraic Methods in Programming*, 140, 2024.

[Heh93] E C R Hehner. *A Practical Theory of Programming*. Springer Verlag, 1993. Latest version available on-line.

[Mil01] J Miller. An historical account of set-theoretic antinomies caused by the axiom of abstraction. June 2001. Available via ResearchGate.

Appendix. Cantor's Theorem for powersets

The set of subsets of S, known as the powerset of S and written as $\mathbb{P}(S)$ is strictly larger than S.

Our proof covers both finite and infinite sets, and is achieved by showing there is no bijection between S and $\mathbb{P}(S)$.

Corresponding to any $f \in S \to \mathbb{P}(S)$ we construct a set $D \subseteq S$ which is different from any set in the range of f.

Since a bijection from S to $\mathbb{P}(S)$ requires that f maps to every point in $\mathbb{P}(S)$ this will ensure that f is not a bijection.

We define $D \mathrel{\widehat{=}} \{ x \mid x \in S \wedge x \notin f(x) \}$

Then, as required, at any $x_0 \in S$

$x_0 \in D \Leftrightarrow x_0 \notin f(x)$ and thus $f(x_0) \neq D$. $\square$

This theorem is related to Cantor's paradox of the universal set U. The universal set must be the largest set since it contains all sets. However $\mathbb{P}(U)$ is larger than U. Since we cannot have any set which is larger than the universal set, the universal set cannot exist.

Implementing Exceptions in Z79Forth/A

François Laagel, IEEE

Introduction to Z79Forth

- Hardware (HD6309 based SBC) and software platform for learning Forth and hardware experimentation.
- HW design started at the end of 2018 and lasted about 3 months.
- Forth is the OS and mostly EEPROM resident.
- Code originally written for the Z80 in 1983 (EDTASM+ by Microsoft).
- Either a 79-STANDARD sub-set or an extended ANS94 core word set.
- 32 KB SRAM, 8 KB EEPROM, mass storage over CompactFlash (64 MB).
- Asynchronous serial line for the console (USB or RS232 based).
- USB powered.
- Has been ported to other platforms (Scott Vitale, David J. Richards).

Z79Forth/A as a Minimalist Implementation

- Produces machine code directly (no VM).
- Return addresses are stored in the 6309 system stack stack (via **Sreg**).
- The 6309 **Ureg** points directly to the data stack.
- Error handler can perform *introspection*.
- No multi-tasking support (single user variable set).
- No files, vocabularies or floating point support.
- Has a dedicated control flow stack.
- Has a dedicated exception stack.

Methodology

Confidence is what you have before you understand the problem. *Woody Allen*

- Prototyping user level exceptions in Forth.
- Design with Gemini (concepts identification, exceptions typology).
- Step by step implementation (6309 assembly).
- Observability concerns.
- Testing/troubleshooting and standard conformance.
- Application level verification.
- Packaging, documentation changes and source control updates.
- Public release on Github.

Constraints Limited EEPROM capacity.

Goals Standard conformance, quality/predictability.

Forth Standard Exception Concepts

- The Exception word set: **CATCH**, **THROW**, **ABORT** and **ABORT"**
- Some incidence on **EVALUATE** and **LOAD**.
- Exception numbers have the same size as a *cell*.
- Negative exceptions are mostly *system generated*.
- Positive exceptions are *application defined*.
- An *error condition* is signalled by throwing an exception.
- Can be nested, so an *exception stack* is needed.
- Entries on the exception stack are called *frames*.
- An exception frame has all the elements describing a saved *process context*.
- The Forth Standard suggests a *reference implementation*, which is not **normative**.

```
( Exceptions )
variable handler
: catch ( xt -- n )
  fp@ >r sp@ >r handler @ >r rp@ handler ! execute
  r> handler ! rdrop rdrop 0 ;
: throw ( n -- )
  dup if handler @ rp! r> handler !
         r> swap >r sp! drop r> r> fp! else drop then ;
```

Brad Nelson's Reference Implementation (SVFIG)

User Level Exceptions Prototyping (1/2)

- Dedicated exception stack primitives: **>E** and **E>**.

```
\ Pop one cell from the exception stack.
: E> ( -- n )
  ESP @ DUP ESbottom = IF
    ." Estack UDF"  ABORT
  THEN

  DUP @
  SWAP CELL+ ESP ! ;

\ Push one cell to the exception stack.
: >E ( n -- )
  ESP @ DUP EStop = IF
    ." Estack OVF"  ABORT
  THEN

  1 CELLS - DUP ESP ! ! ;
```

- A stack frame has 3 cells: the system stack pointer, the data stack pointer and the return stack pointer.

User Level Exceptions Prototyping (2/2)

```
: CATCH ( xt -- n | 0 )   \ Build up exception frame.
  s CELL+ >E              \ Save S register ('Sreg0')
  sp@ CELL+ >E            \ Data stack pointer backup ('sp0')
                          \ We omit the currently stacked 'xt'
  rp@ >E

  EXECUTE

  \ Normal execution in the absence of exceptions. We have to
  \ assume that return stack integrity rules are followed.
  E> DROP                 \ Drop rp@
  E> DROP                 \ Drop sp@
  E> DROP                 \ Drop saved Sreg
  0 ;                     \ "No problemo" indication
```

```
: THROW ( n -- )
  ?DUP IF                 \ An exception was triggered
    \ Restore saved context from the exception stack.
    E> rp!                \ Restore the return stack pointer

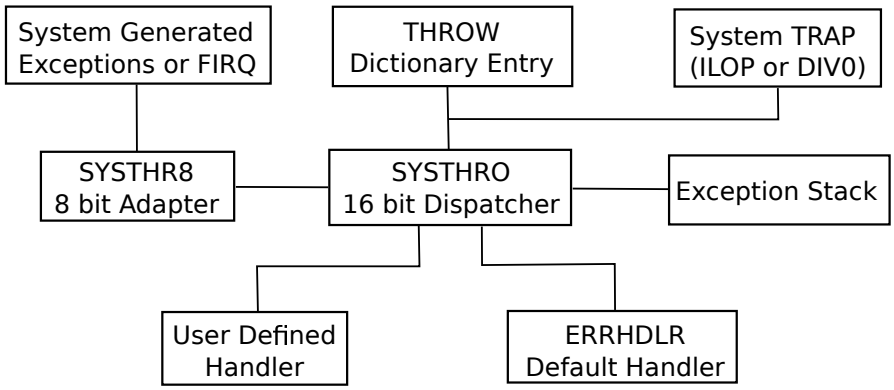
    >R                    \ E: Sreg0\sp0, R: n
    E> sp!                \ Restore the data stack pointer
    R>                    \ Stack up 'n' to the data stack
    E>                    \ E: -, S: n\Sreg0
    \ Need to restore the S register from the TOS and then
    \ simply RTS like a good boy. Using inline code.
    [ jsrNPOP,
      $1F14 , ]           \ TFR X,S
  THEN ;                  \ Implied RTS sitting right here.
```

Implementation–Taming the Error Handler

- An 8 bit affair. *Abstraction* needed.
- A cell based interface was required for **THROW**.
- Negative error codes handling capability.
- Need to align error codes to standard specified exception numbers.

```
* Error condition codes passed through Breg to SYSTHR8.
EABORT  equ     -1      STD -- ABORT
EABRTQ  equ     -2      STD -- ABORT"
EDSOVF  equ     -3      STD -- DS overflow
EDSUDF  equ     -4      STD -- DS underflow
ERSOVF  equ     -5      STD -- RS overflow
ERSUDF  equ     -6      STD -- RS underflow
ESSEGV  equ     -9      STD -- Illegal argument (CHAR FIND PICK EXECUTE)
EOORNG  equ     -11     STD -- OoR error
EUNDEF  equ     -13     STD -- ? / Undefined word
ESTATE  equ     -14     STD -- Incorrect STATE
ENONAM  equ     -16     STD -- Missing word (POSTPONE : CREATE)
ENAMLN  equ     -19     STD -- Name too long
ERONLY  equ     -20     STD -- RO word
ENOSUP  equ     -21     STD -- IO error or TRAP (DIV0 ILOP)
EILCST  equ     -22     STD -- Illegal construct
EUINTR  equ     -28     STD -- SIGINT
ENOCRE  equ     -31     STD -- Not CREATEd
* Allocated against the standard specification. These are not catchable--i.e.
* we branch directly to ERRHDLR (under the 16 bit interface). They're not
* recoverable anyway.
EINOOR  equ     -32     >IN OoR
EESOVF  equ     -33     ES overflow
```

Exception Dispatching



- The error handler becomes a *backend* invoked only for uncaught exceptions.
- We can have a cell based interface and keep the old 8 bit API.

Actual Implementation–CATCH

```
CATCH   fcb     $5              ANSI (Exception)
        fcc     'CATCH'         ( i*x xt -- j*x 0 | i*x n )
        fdb     THROW
        RFCS
        jsr     MIN1PST         Make sure at least 'xt' is stacked up
* Save exception frame.
        orcc    #FFLAG          Disable FIRQ
        leax    2,u
        bsr     EPUSH           sp@ CELL+ >E (omit 'xt')
        ldx     RSP
        bsr     EPUSH           rp@ >E
        tfr     s,x             Return address pointer if uncaught
        bsr     EPUSH           Sreg >E
        andcc   #^FFLAG         Re-enable FIRQ
        RFXT    jsr,EXCT+10     xt for EXECUTE
* Normal execution in the absence of exceptions.
        orcc    #FFLAG          Disable FIRQ
        bsr     EPOP            E> DROP (original Sreg)
        bsr     EPOP            E> DROP (original rp@)
        bsr     EPOP            E> DROP (original sp@)
        andcc   #^FFLAG         Re-enable FIRQ
        tfr     0,x
        jmp     NPUSH           "No problemo" indication to DS
```

Actual Implementation–THROW

```
THROW   fcb     5               ANSI (Exception)
        fcc     'THROW'         ( k*x n -- k*x | i*x n )
        fdb     EBUFS
        RFCS
        jsr     NPOP
        tfr     x,d
        tstd                    0 exception number?
        bne     SYSTHRO         if not, we continue execution in SYSTHRO
        rts                     just EXIT if 'n' is 0
▉
* Compatibility entry point for 8 bit system generated exceptions.
* Note: Breg never is zero if we enter via this inferface.
SYSTHR8 sex                     Sign extension Breg to Dreg

* Exception number is in D. Route it to ERRHDLR or to the user code just
* following the latest call to CATCH, depending on the state of the exception
* stack. Preserve X and D before resorting to ERRHDLR (unhandled exception).
SYSTHRO pshs    x
        ldx     ESP             The exception stack pointer
        cmpx    #ESTBOT
        puls    x               This does not affect CCreg
        lbeq    ERRHDLR         Stack is empty, invoke the default handler
* At this point we need to return control just after the latest call to CATCH.
        orcc    #FFLAG          Disable FIRQ
        bsr     EPOP            Does not affect Dreg or Yreg
        tfr     x,s             Restore Sreg
        bsr     EPOP
        stx     RSP             E> rp!
        bsr     EPOP
        tfr     x,u             E> sp!
        andcc   #^FFLAG         Re-enable FIRQ
        tfr     d,x
        jmp     NPUSH           Exception number to the data stack
```

ABORT" Concerns (1/2)

Run-time:

(*i* * *x* *x*¹ -- | *i* * *x*) (R: *j* * *x* -- | *j* * *x*)

Remove *x*¹ from the stack. If any bit of *x*¹ is not zero, perform the function of `-2` THROW, displaying *ccc* if there is no exception frame on the exception stack.

The Standard Specification

- If TOS in NZ, a string (**caddr\length**) is pushed on the data stack and -2 is thrown.
- If that exception is caught, it is supposed to be handled by the application.
- It is uncaught, the string is displayed and the semantics of ABORT is performed.

```
: ABORT"                    \ ANSI (Exception)
  ( Compilation: "ccc<quote>" -- )
  ( Runtime: i*x x1 --  | i*x ) ( R: j*x --  | j*x )
  POSTPONE IF
    POSTPONE S"
    -2 POSTPONE LITERAL
    POSTPONE THROW
  POSTPONE THEN ; IMMEDIATE RESTRICT
```

The Z79Forth/A Implementation

ABORT" Concerns (2/2)

If ABORT" is caught and there was something on the data stack, there will still be something (same depth + 1) on the data stack but there is no way for a user defined handler to tell, in a predictable manner, what it is...

```
Gforth 0.7.9_20231019
Authors: Anton Ertl, Bernd Paysan, Jens Wilke et al., for more type `authors'
Copyright © 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>
Gforth comes with ABSOLUTELY NO WARRANTY; for details type `license'
Type `help' for basic help
: foo ABORT" Gottya!" ;  ok
1 ' foo CATCH DROP HEX .S <1> -2  ok 1

b=16|<1> #-2                                |o:Forth Forth Root     Forth


Z79Forth/AE 6309 ANS Forth System
20260820 (C) Francois Laagel 2019

RAM OK: 32 KB
SanDisk SDCFJ-128
Block #1 loaded
: foo ABORT" Gottya!" ;    ok
1 ' foo CATCH DROP HEX .S  19DD  ok
$19DD .' FOO+000C ok
see foo
19D1 BDE817     jsr     NPOP
19D4 2603       bne     *+05
19D6 7E19F6     jmp     FOO+0025
19D9 7E19E4     jmp     FOO+0013
19DC 07         fcb     $07
19DD 47         fcb     $47
19DE 6F         fcb     $6F
19DF 74         fcb     $74
19E0 74         fcb     $74
19E1 79         fcb     $79
19E2 61         fcb     $61
19E3 21         fcb     $21
19E4 8E19DC     lxi     FOO+000B
19E7 BDE808     jsr     NPUSH
19EA BDF7E2     jsr     COUNT
19ED 8EFFFE     lxi     ???
19F0 BDE808     jsr     NPUSH
19F3 BDEA5B     jsr     THROW
19F6 39         rts ok

CTRL-A Z for help | 38400 8N1 | NOR | Minicom 2.8 | VT102 | Offline | ttyS0
```

A Case for **EVALUATE** (1/2)

```
: EVALUATE ( i*x c-addr u -- j*x )
   SAVE-INPUT N>R
   ( c-addr u) #TIB 2!  >IN OFF  LINE OFF  BLK OFF  -1 'SOURCE-ID !
   ['] INTERPRET CATCH  NR> RESTORE-INPUT DROP  THROW ;
```

SwiftForth x86_64 4.1.10 Definition

A Case for **EVALUATE** (2/2)

```
LOAD    fcb     4               ANSI (Block)
        fcc     'LOAD'          ( i*x u -- j*x )
        fdb     TERPRET
        RFCS
        jsr     NPOP            ZFLAG is set by NPOP
        bne     LOAD1
        rts                     Block 0 is _not_ loadable
LOAD1   jsr     SAVINP          SAVE-INPUT. X is preserved
        stx     UBLK            Update BLK with the LOAD argument
        ldd     #BLKSIZ         1024 bytes
        std     ISLEN           Set input stream length
        clrd
        sta     SRCID           Not invoked in EVALUATE context
LOAD2   std     UTOIN           Clear >IN
        std     ISEADDR         End of input stream address (excluded)
* Map the new BLK in, interpret code from there.
        RFXT    ldx,#TERPRET+12 XT for INTERPRET
        jsr     NPUSH           to the data stack
        RFXT    jsr,CATCH+8     ['] INTERPRET CATCH
        jsr     RSTINP          RESTORE-INPUT
        jsr     BKIN2PT         Map BLK in (if needed) and update BSBFADR
* We need to throw back the caught exception (if any) before returning.
        RFXT    jmp,THROW+8

EVAL    fcb     8               ANSI (Core/Block)
        fcc     'EVALUATE'      ( i * x c-addr u -- j * x )
        fdb     LOAD
        RFCS
        jsr     MIN2PST         Need at least 2 parameters on the data stack
* The whole thing looks like LOAD1 except we do not need to map a block in.
        jsr     SAVINP          SAVE-INPUT. X and Y are preserved
        UCNPOP                  U to X
        stx     ISLEN           Set ISLEN from U
        UCNPOP                  C-ADDR to X
        stx     BSBFADR         Set BSBFADR from C-ADDR
        lda     #$FF
        sta     SRCID           -1 (byte) to SRCID. Invoked in EVALUATE context
        clrd                    Future value for >IN
        std     UBLK            Target block number is zero
        bra     LOAD2           Interpret, restore input context and proceed
```

Z79Forth/A Assembly Dictionary Entries

Observability Notes

- Serial line input is interrupt driven in Z79Forth. When **^C** is detected (in the FIRQ handler), control will redirected to a small piece of code which:
  - Sets the return address to SYSTHRO.
  - Preserves the original user code return address in the system stack.
  - Sets the exception number to EUINTR (via Dreg).
- The perils of tail call optimization. **CATCH** should not be allowed to be terminal. A few changes to **;** were needed.

```
* Optimization: replace the last JSR by a JMP, if possible.
        leay    -3,x            Yreg has HERE - 3, Dreg has JSRLAST
        cmpr    d,y
        bne     @rtsreq
        ldd     1,y             Address of the last word called
        RFXT    cmpd,#CATCH+8   Observability: we definitely do
        beq     @rtsreq         not want CATCH to be terminal
* Tail call optimization applies (Case #3).
        lda     #JMPOPC
        sta     ,y
        bra     @finalz
@rtsreq lda     #RTSOPC         RTS inherent
        sta     ,x+
@finalz stx     DICEND          Update HERE
        IFNE    RELFEAT
        RFXT    jsr,MONITOR+10  XT for MONITOR. All : words are candidates
*                               for integrity check by ICHECK.
        ENDC                    RELFEAT
        rts
```

Conclusions

- Exception support in Forth is very useful and can greatly simplify your code.
- An application should not hesitate to re-throw an exception that it does not own. The default handler might provide more useful information.
- Firmware supported exceptions now is an integral part of Z79Forth/AE.
- Standard conformance is essential.

Q&A

Gforth 1.0

M. Anton Ertl
TU Wien

Numbers

- 18 years (last release 2008)
  34 years since Gforth started
- 7837 non-merge commits
- documented words: +767 -31; total: 1545

Documented (in)stability

- `gforth-0.2` ... `gforth-1.0`: stable
- `gforth-experimental`: May change (mostly due to external factors)
- `gforth-internal`: May change at will
- `gforth-obsolete`: Intent to delete

Development: IDE features

- IDE features: `locate`, `where`, source for backtrace entries
- EuroForth 2020
- switch between `where` or backtrace entries with cursor keys
- `browse` (`locate` words matching a pattern)

Development: Show and input data and code

- Status line shows stacks, base, search order
- `...` prints stack items as:
- `‘word ‘‘word <word> <word+$10> "string" <dest> <colon-sys>`
  and the red ones can be input
- `6k5 6.5k 23%`
- `->v +>v ’v @>d =>d`
- `forth:and assembler:and`
- `${LANG}`
- extend the text interpreter with additional recognizers
  EuroForth 2012, 2020

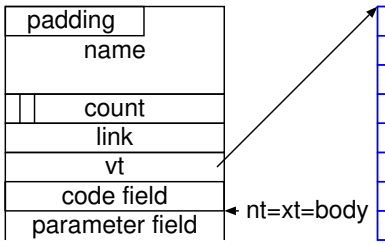
Multi-Tasking

- Based on traditional words, but:
- Multiple hardware threads and cores
  No implicit atomic sequences for communicating between tasks
- Explicit mutexes
- Words for atomic memory updates (for lock-free algorithms)
- Message queues (actor model variant)
- Distribute work among cores with Cilk

Miscellaneous

- C interface: callbacks into Forth code and other enhancements
- Sections: multiple dictionary pointers (EuroForth 2016)
  just one section stack plus named recognizers
- Postpone mode: `]] if 5 then [[`
- Value/defer-flavoured fields and per-task (user) values/defers
  defer-flavoured locals (`xt: d`) and local buffers (`buf[ 10 ]`)
  Add your own with user-defined `to` etc.
  `addressable: addr`
- Closures (Euroforth 2018)
  Pure-stack closures, one-shot closures
  `( n ) [n:h1 + ;] \ xt ( n1 -- n1+n )`
- Interactive multiline `>R R>` (EuroForth 2019)
- ABI-Code for assembler portability between engines (EuroForth 2010)
- Improved decompiler (EuroForth 2024)
- HTML documentation: words link to their documentation

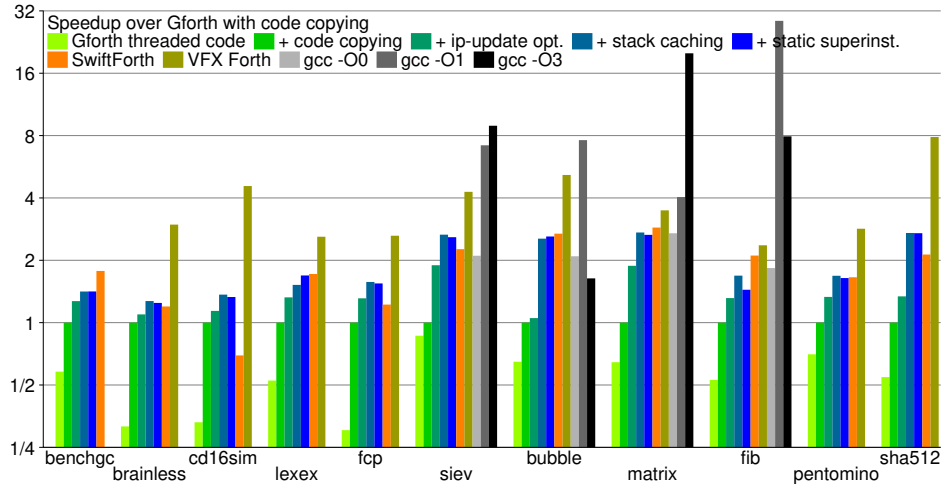
New header layout



- EuroForth 2019
- xt = nt = body address
- `create-from` prototype
- flags for
  `compile-only`
  `obsolete`
- `synonym`
- `forward`

7

Performance



- IP-update optimization (ECOOP 2024)
- literal stack

8

Conclusion and Status

- Many years
- Many changes
- Currently still working on portability
- If you can build a current snapshot, you have Gforth 1.0

9

Control Structure Enhancements

M. Anton Ertl
TU Wien

Generalized Eaker's case

```
: collatz ( u -- )
    case
        dup .
        1 of endof
        dup 1 and ?of 3 * 1+ contof
        2/
    next-case ;
```

- ?of 13 uses
- next-case 2 uses
- contof 19 uses

Walking across arrays

```
create v 1 , 3 , 7 ,
: foo
  v 3 cell array>mem ( addr bytes stride ) MEM-DO
    i @ .
  LOOP ;
```

- array>mem 2 uses
- mem-do 5 uses
- mem+do 15 uses

Floating-point output

M. Anton Ertl
TU Wien

Why not just use existing words?

```
0.1e          f. 0.1                    output without "e" (double-cell)
0.1e         fs. 1.00000000000000E-1    non-zero exponent
0.1e         fe. 100.000000000000E-3    non-zero exponent
0.1e 7 5 3 f.rdp 0.10000                output without "e" (double-cell)
```

- Result valid as FP input: `fs. fe.`
- ~~Result converted to FP produces original value~~
- Output not overly long: `fs. fe. f.rdp`
- ~~output exact (no rounding)~~
- Useful in some contexts, but not in others

E.EXACT

```
0.1e          e.exact .1000000000000000055511151231257827021181583404541015625e
0.1e 0.1e f*  e.exact .0100000000000000019428902930940239457413554191589355468 75e
1e8           e.exact 100000000e
1e-8 e.exact  1.0000000000000000209225608301284726753266340892878361046314239501953125e-8
0.999999e     e.exact .99999899999999997124433548378874547779560089111328125e
```

- Result valid as FP input
- Result converted to FP produces original value
- ~~output not overly long~~
- output exact (no rounding)
- Useful for demonstrating FP properties
- based on exact `represent`/`ecvt_r()`

E.

```
0.1e          e. .1e
0.1e 0.1e f*  e. .010000000000000002e
1e8           e. 100000000e
1e-8          e. 1e-8
0.999999e     e. .999999e
```

- Result valid as FP input
- Result converted to FP produces original value
- output not overly long
- ~~output exact (no rounding)~~
- Useful as default FP output word

E.P

```
0.1e          5 e.p .1e
0.1e 0.1e f*  5 e.p .01e
1e8           5 e.p 1e8
1e-8          5 e.p 1e-8
0.999999e     5 e.p 1e
```

- Result valid as FP input
- ~~Result converted to FP produces original value~~
- output ≤ `e.` (with $p < 16$)
- ~~output exact (no rounding)~~
- Useful if `e.` output is too long

[4]

Security and ARM64

Consequences of macOS security on Apple Silicon and …

Stephen Pelc, stephen@vfxforth.com, Wodni & Pelc GmbH, Sept 2026

In the beginning …

Forth and a modern “secure” operating system

- A classical Forth for an O/S reads, writes or executes a memory block (RWX)
- You can define a macho segment as RWX, but …
  real memory is R_W or R_X only - system will crash
  O/S functions to switch between R_W and R_X
- Address randomisation cannot (should not) be turned off
- Position independent code and hence position independent data
- There are many instructions in the ARM64 ISA for security

Dealing with a modern “secure” operating system

This reminds us of embedded systems in Forth?

- Embedded systems
  Flash is R_X, RAM is RW_ (and maybe RWX)
- Look at unadopted cross compiler proposal
  `CREATE` and **`,`** affected by **`CDATA`** (.text), **`IDATA`** (.data) and **`UDATA`** (.bss)
  See: **`https://www.mpeforth.com/resource-links/downloads`**
- The cache is a pain unless the O/S has sufficient support routines
- To minimise switching overhead, must separate code and data

Solutions

Separate code and date

- Position independent code, position independent data
  This is the C model. Sections are relocated at program load.
  Relocation data is part of the binary image.
- Load the Forth image twice, once for code, once for data.
  Sacrifice two registers as base pointers for code and data.
  Used by one Forth system
- Generate a relocation bitmap for code and data and apply it at load time.
  Relatively easy for code, more complex for data.
  Based on existing VFX code for generating DLLs.
- Ignore macOS

Benefits and sacrifices

Is it worth saving two registers, source compatibility and complexity?

- Will the macOS security decisions be taken up by other systems?
- The mood (with no evidence) suggests that it will.
- Even with 31 registers, losing two will reduce performance in the long term.

VFX register usage

The pain begins

- VFX use

=======			
Integer/GP			

SP/X30	SP	System stack pointer - 16 byte aligned	
X30	LR	Link register	
X29	FP	PSP - Forth parameter/data stack pointer ; 8 byte aligned	
X28		RSP - Forth return stack pointer - 8 byte aligned	
X27		UP - user area pointer	
X26		TOS - cached top of data stack	
X25		LP - locals pointer	
X24		FSP - float stack pointer	
X23		Index for DO/LOOP	non-volatile across ABIs
X22		Index~Limit	non-volatile across ABIs
X21..X19		Return stack cache, 3 items	
X18		Platform register - do not use	
X17..X8		VFX data stack, 10 items	
X0..X7		scratch and system in/out regs	

- Float

V0..V7	FP and SIMD scratch and system in/out
D8	FTOS - cached top of FP stack

Tools - 1

What we have

- Cross compiler - x64 -> ARM64 - done
  Can use Rosetta under macOS and Linux
  Rosetta has bugs which can destroy integrity of the target image
  First bytes of new 4k page can be corrupt
- Cross compiler ARM64 -> ARM64 - done
  Linux build works fine on Raspberry Pi and Debian ARM64 in Mac VM

Tools - 2

What we need to do

- Redesign kernel to accept **`CDATA`** **`IDATA`** and **`UDATA`**
  `UDATA` can be avoided if **`BUFFER:`** always allocates from the heap.
- Ensure compatibility across all CPUs with and without separation
- Implement code and data relocation code
- Implement startup loader code
  macOS has several additional requirements that are undocumented.
  macOS has documented requirements that make a simple loader more difficult.
- Implement **`SAVE`**

Thanks

- Questions?